Data Representation and Compression

Announcements

- The first lab exam is tonight, during the lab session.
  - You may use your own computer
- PA6 due tonight
- PA 7, PS 7 and Lab 8 on July 24th

Today: Data Compression

- Data Compression
  - Lossless vs lossy
- Measuring Information
  - Algorithmic Information Theory
  - Shannon's Information Theory
- Data Compression: Encoding
- Data Compression: Decoding
- Huffman Coding
- Parity Bits

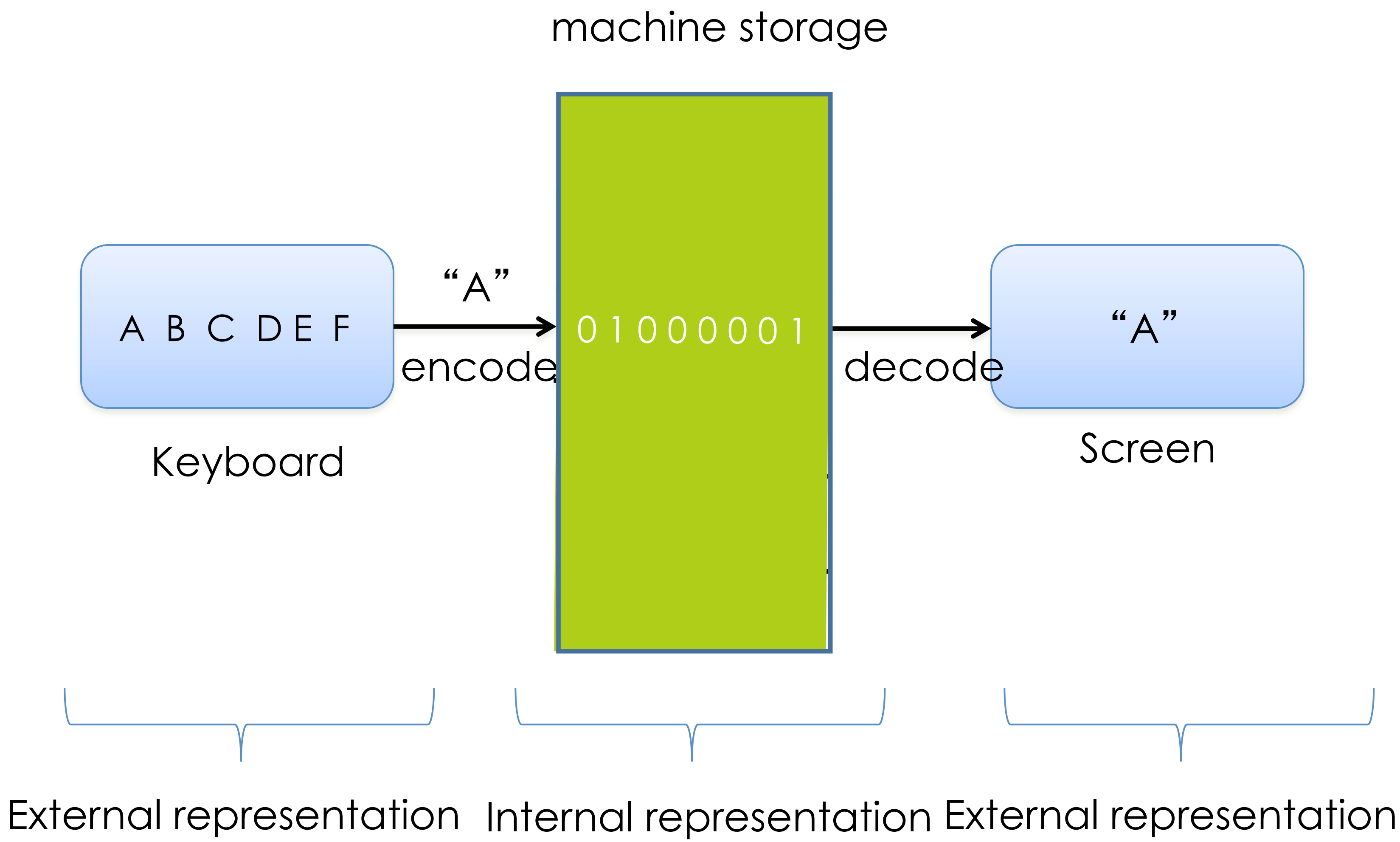
Review:

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

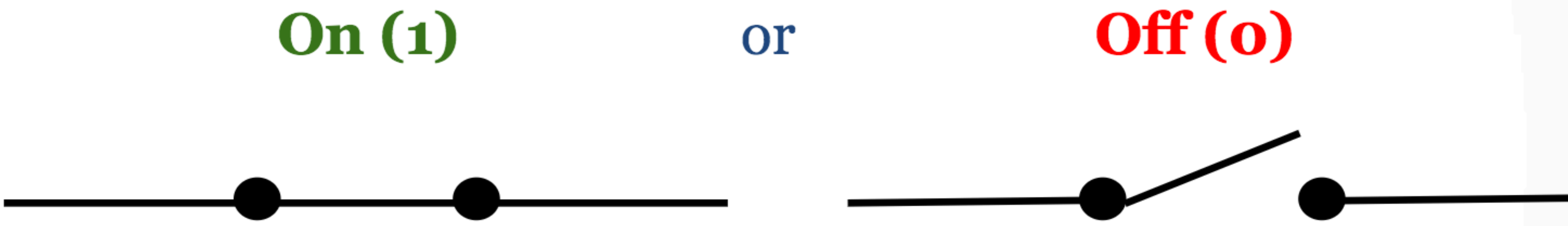


Computers speak in binary

- Binary: A pair of opposites
  - On or Off
  - Yes or No
  - 0 and 1

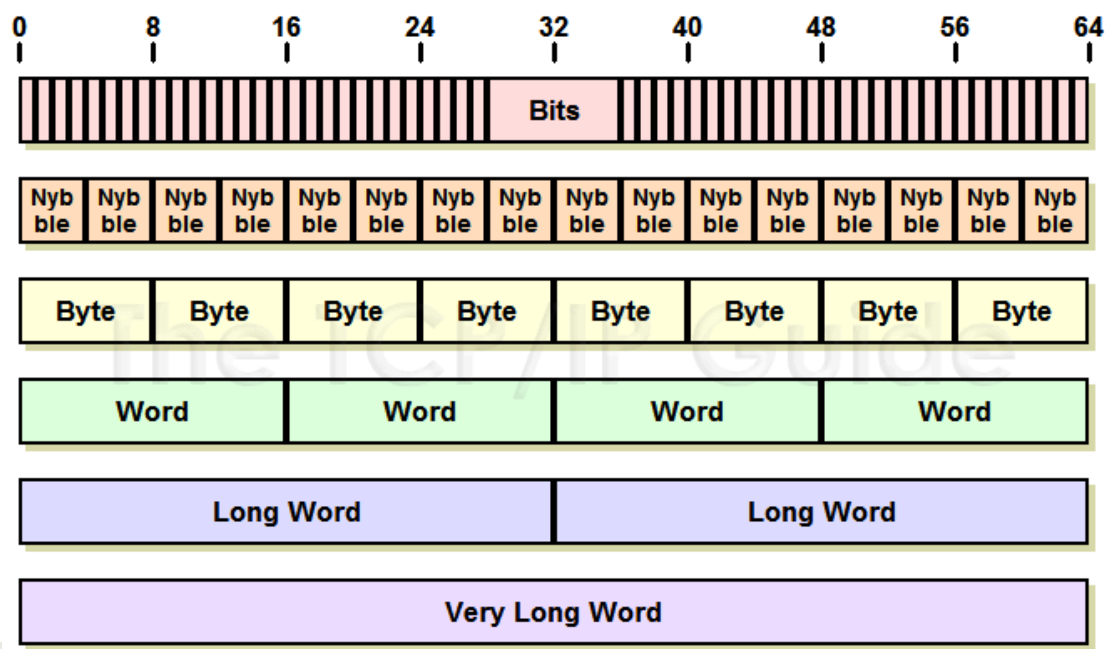
Computers speak in binary

- Binary: A pair of opposites
  - On or Off
  - Yes or No
  - 0 and 1
- Where does binary come from?
  - Computers are powered by electricity
  - Electricity either goes through or doesn't go through a wire

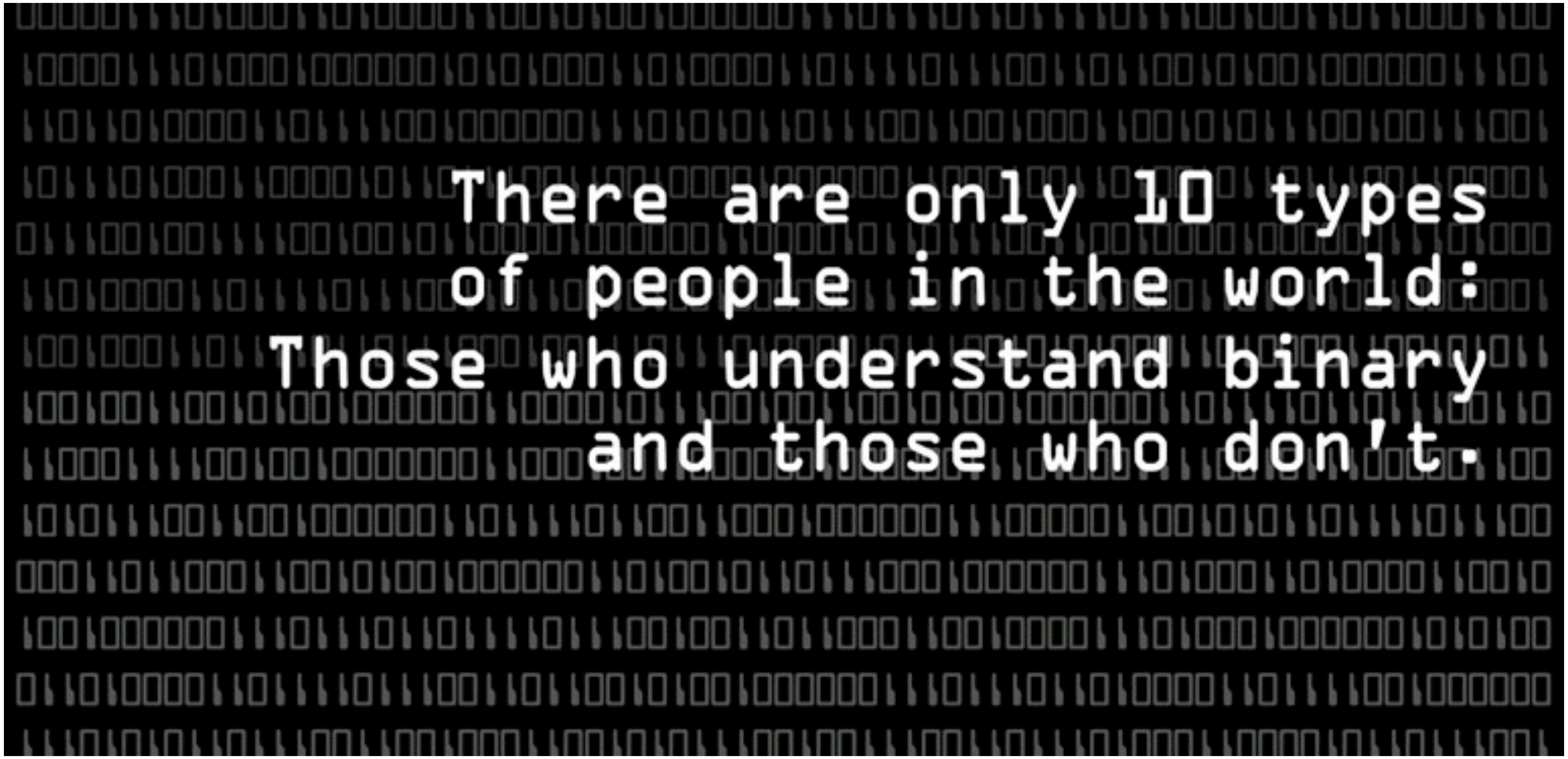


Machine storage

- Bit == the smallest piece of information the computer can store
  - 0's and 1's represent the bits
- (smallest unit) 1 byte = 8 bits
- (biggest chunk) 1 word = 16, 32 or 64 bits (depending on your machine)
- Machine storage capacity is expressed as bytes and words



Too many jokes..



Representing Non-negative (unsigned) integers

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

Encoding algorithm:
convert quantity to a given base

- 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

Encoding algorithm: convert quantity to a given base

- To represent n=6 with **b=2**
  - 6 // 2 = 3, **r=0**
  - 3 // 2 = 1, **r=1**
  - 1//2 = 0, **r=1**

Binary numeral: 110

What it means:
$0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 =$ "six"

- To represent n=2019 with **b=10**
  - 2019//10, r=9
  - 201//10, r=1
  - 20//10, r=0
  - 2//10, r=2

Decimal numeral: 2019

What it means:
$9 \times 10^0 + 1 \times 10^1 + 0 \times 10^2 + 2 \times 10^3 =$
"two thousand and nineteen"

Decoding algorithm for **unsigned (non-negative) integers** (decode 1010)

- **Binary numeral: 1010**
- What it means:

$0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3$

$= 10$

Encoding Algorithm for **unsigned (non-negative) integers** (encode 6)

To represent n=6 with **b=2**

- **6 // 2 = 3, r=0**
- **3 // 2 = 1, r=1**
- **1//2 = 0, r=1**

Binary numeral: 110

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

+	0	1
0	0	1
1	1	10

- 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
```

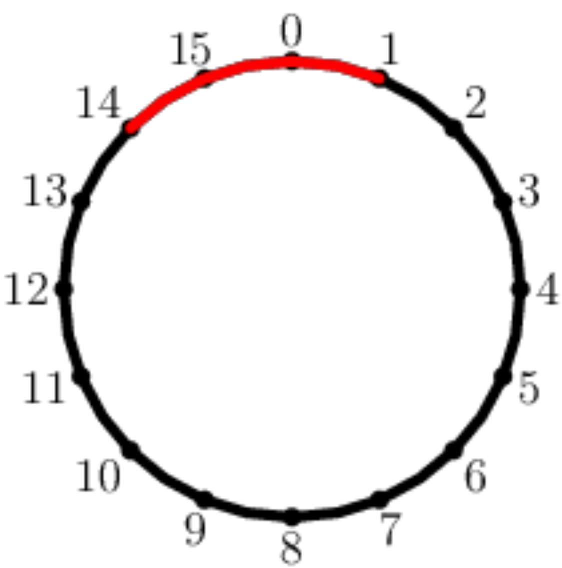
- The machine retains only 0100 (the "least significant" bits), so $(n+n) - n$ **not** always equal to $n + (n - n)$

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 (20 mod 16)

overflow can be ignored or signaled as an error



Representing Negative (signed) 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…

Two's complement is an approach for representing **negative integers**

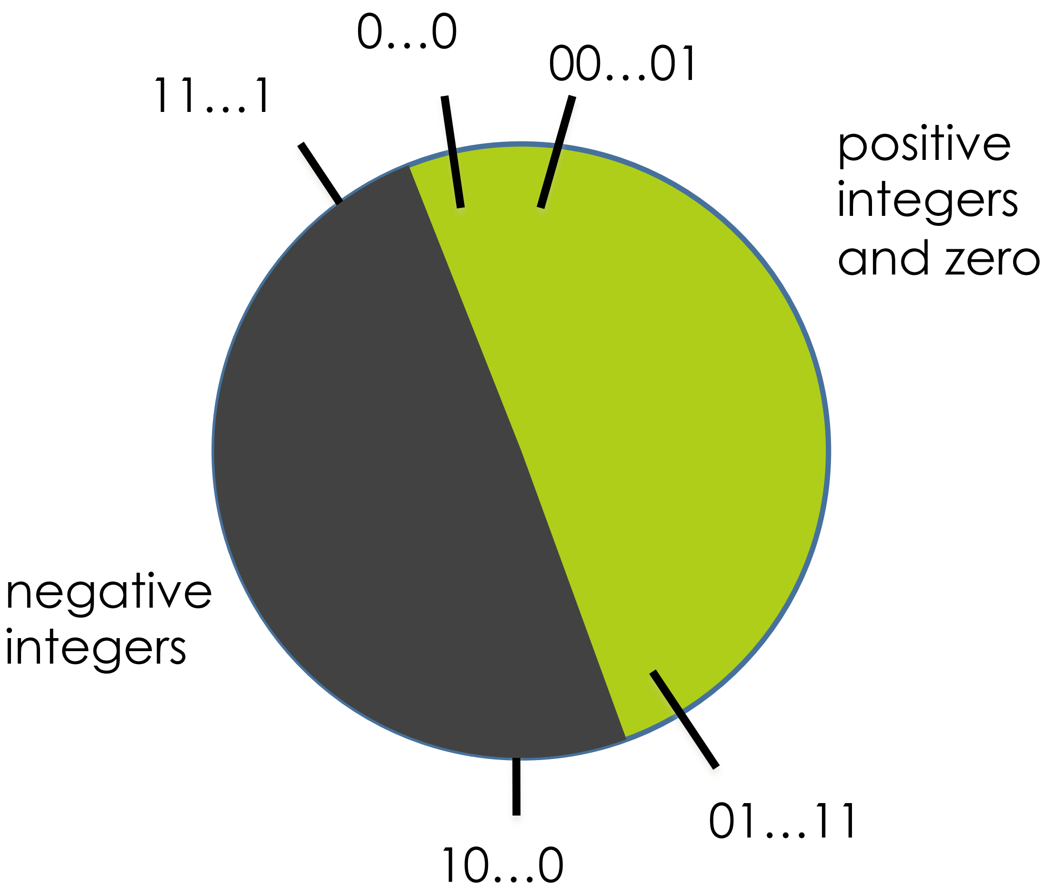
Decoding Algorithm for **negative integers** (decode 1010, 4 bits)

- **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 (in base 2!!)
    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

Encoding Algorithm for **negative integers** (encode -52 in 8 bits)

- Start by encoding +52
  - **52 = 00110100**
- Flip each bit (one's complement):
  - flip **00110100** to get **11001011**
- Add 00000001
  - **11001011 +** 00000001
  - **= 11001100**
  - = −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

Negative vs Non-negative (Signed vs Unsigned) integers

k bits can represent 2^k **different things**!

k bits can represent 2^k **different things**!

Representing
non-negative (unsigned) integers

Representing
negative (signed) integers
(two's complement)

k bits can represent 2^k **different things**!

Representing
non-negative (unsigned) integers

- *k* bits can represent 2^k things
  - For k = 3, $2^3 = 8$

Representing
negative (signed) integers
(two's complement)

- *k* bits can represent 2^k things
  - For k = 3, $2^3 = 8$

k bits can represent 2^k **different things**!

Representing non-negative (unsigned) integers

- k bits can represent 2^k things
  - For $k = 3$, $2^3 = 8$
- Represent <u>non-negative integers</u>

$0 \ldots 2^k - 1$

For $k = 3$: 0,1,2..., 6, 7

Representing negative (signed) integers (two's complement)

- k bits can represent 2^k things
  - For $k = 3$, $2^3 = 8$
- Represent <u>negative and non-negative integers</u>

$-2^{k-1} \ldots +2^{k-1} - 1$

For k = 3: -4, ..., 0, 3

k bits can represent 2^k **different things**!

Representing **non-negative (unsigned) integers**

- k bits can represent 2^k things
  - For k = 3, $2^3 = 8$
- Represent non-negative integers

$0 \ldots 2^k$-1
For k = 3: 0,1,2..., 6, 7

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

Representing **negative (signed) integers** (two's complement)

- k bits can represent 2^k things
  - For k = 3, $2^3 = 8$
- Represent negative and non-negative integers

$\mathbf{-2^{k-1} \;\ldots\; +2^{k-1}-1}$
For k = 3: -4, ..., 0, 3

000	001	010	011	100	101	110	111
0	1	2	3	-4	-3	-2	-1

k bits can represent 2^k **different things**!

Representing
non-negative (unsigned) integers

- k bits can represent 2^k things
  - For k = 3, $2^3 = 8$
- Represent non-negative integers

0…2^k-1
For k = 3: 0,1,2…, 6, 7

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

Representing
negative (signed) integers
(two's complement)

- k bits can represent 2^k things
  - For k = 3, $2^3 = 8$
- Represent negative and non-negative integers

$\mathbf{-2^{k-1} \ldots +2^{k-1}-1}$
For k = 3: -4, …, 0, 3

000	001	010	011	100	101	110	111
0	1	2	3	-4	-3	-2	-1

k bits can represent 2^k **different things**!

Representing **non-negative (unsigned) integers**

- *k* bits can represent 2^k things
  - For k = 3, $2^3 = 8$
- Represent non-negative integers

  0…2^k-1
  For *k* = 3: 0,1,2…, 6, 7

- Encoding/Decoding
  - Convert to and from base 2

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

Representing **negative (signed) integers** (two's complement)

- *k* bits can represent 2^k things
  - For k = 3, $2^3 = 8$
- Represent negative and non-negative integers

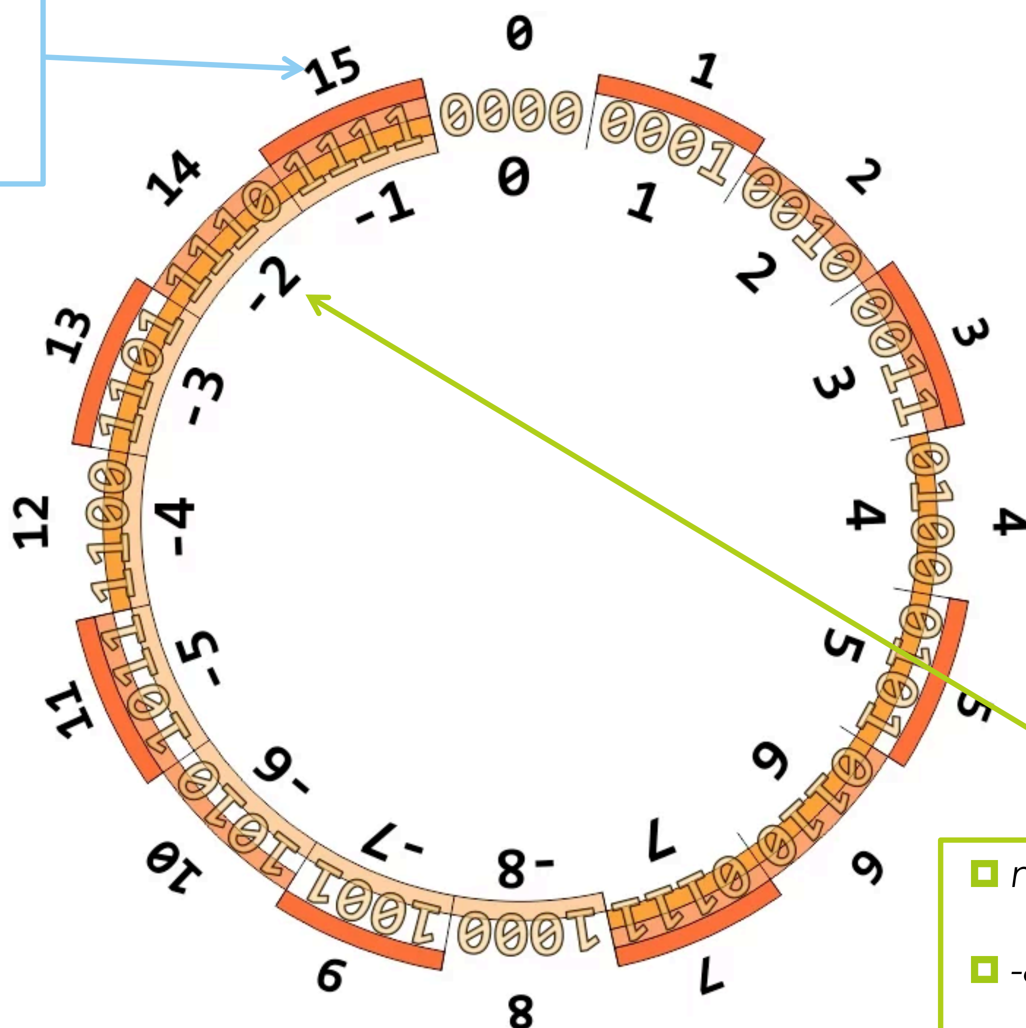
  -2^{k-1} … $+2^{k-1}-1$
  For k = 3: -4, …, 0, 3

- Encoding/decoding
  - If negative, flip, add 1; if positive convert from base 2

000	001	010	011	100	101	110	111
0	1	2	3	-4	-3	-2	-1

For k = 4, binary representation

- *non-negative*
- *0,1,2,3 … 15*
- *outer circle*



- *non-negative + negative*
- *-8,-7, …, 0,6,7*
- *inner circle*

k bits can represent 2^k **different things**!

Representing
non-negative integers

Representing
negative integers
(two's complement)

bits	minimum (0)	maximum (2^k-1)
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)

bits	minimum (-2^{k-1})	maximum ($+2^{k-1}-1$)
8	$-2^7 = -128$	$2^7 - 1 = +127$
16	-2^{15} = –32,768	$2^{15} - 1$ = +32,767
32	-2^{31} = –2,147,483,648	$2^{31} - 1$ = +2,147,483,647
64	-2^{63} = -9,223,372,036,854,775,808	$2^{63} - 1$ = +9,223,372,036,854,775,807

From whole numbers to rational numbers

Rounding in binary

```
>>> x = 1/10
>>> x
0.1
>>> y = 2/10
>>> y
0.2
>>> x + y
0.30000000000000004
```

python prints a rounded value

Ack! Whyyyy?

most decimal fractions cannot be represented exactly as binary fractions!!

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

Similar in decimal to:
1/3 = 0.33333…

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

(1 bit) (23 bits) (8 bits)

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 $.75 = \frac{1}{2} + \frac{1}{4} = 2^{-1} + 2^{-2}$

$$5.75 = 5 + 0.75$$
$$= 101 + 0.11 \text{ (i.e. } 2^{-1} + 2^{-2}\text{)}$$
$$= 101.11 = 1.0111 \times 10^{10}$$

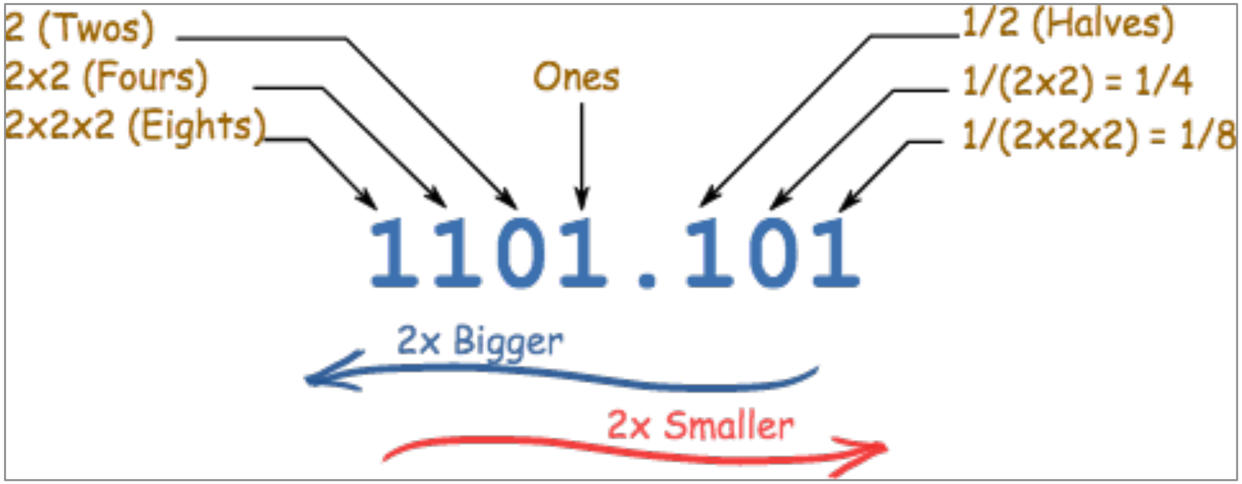
decimal

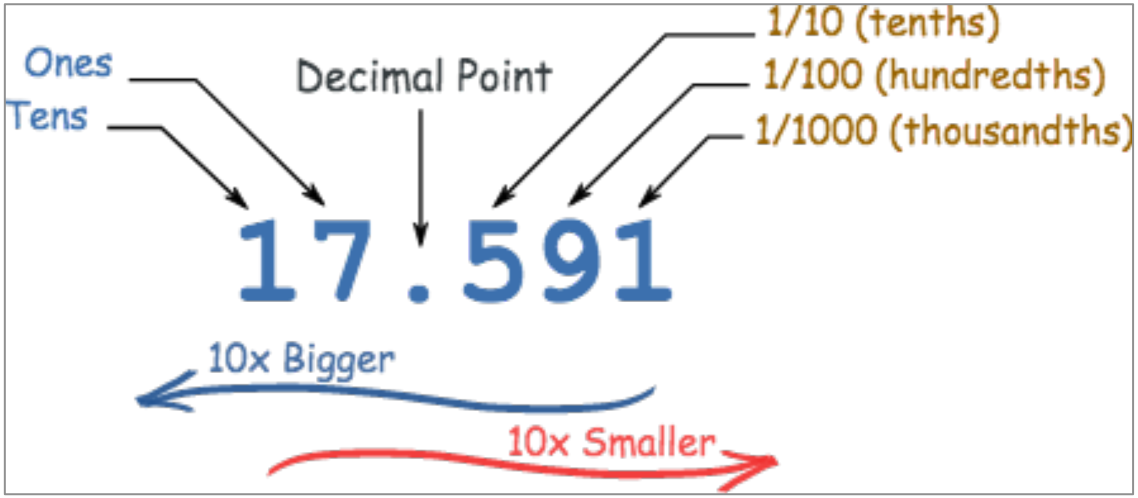
binary

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

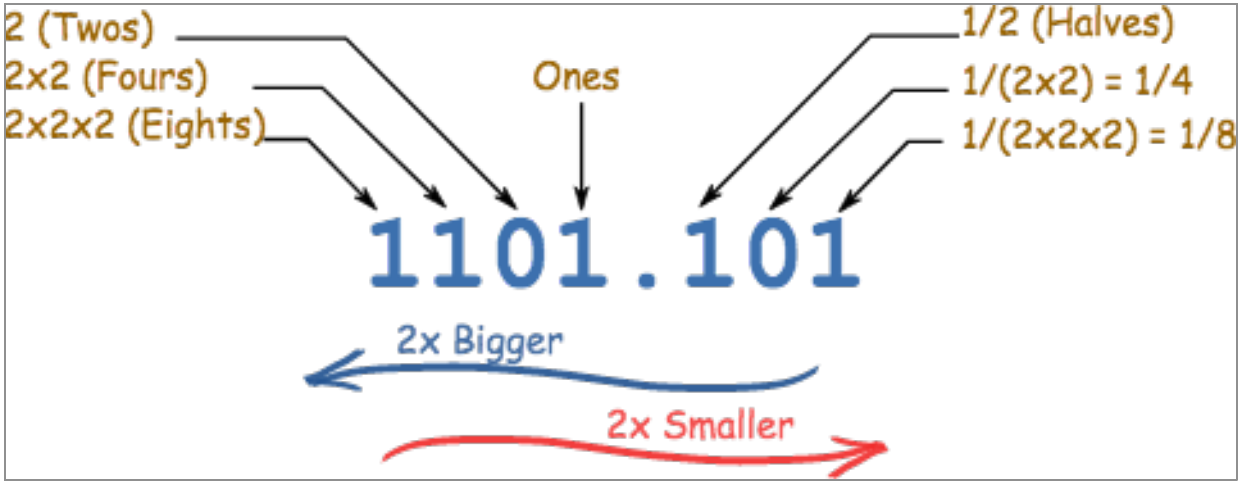
101.11 has *mantissa* 1.0111 and *exponent* 10

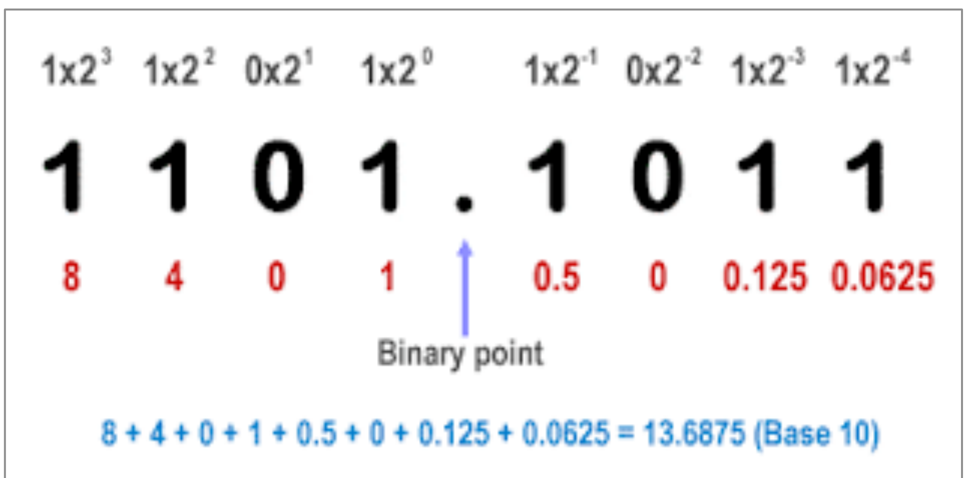
Float point in binary





Float point in binary





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

Compression and decompression

- Reduce storage and for faster transfer of data over networks

- Would like two easily computable functions:

  `compress(m)`

  `decompress(m)`

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

Types of data compression

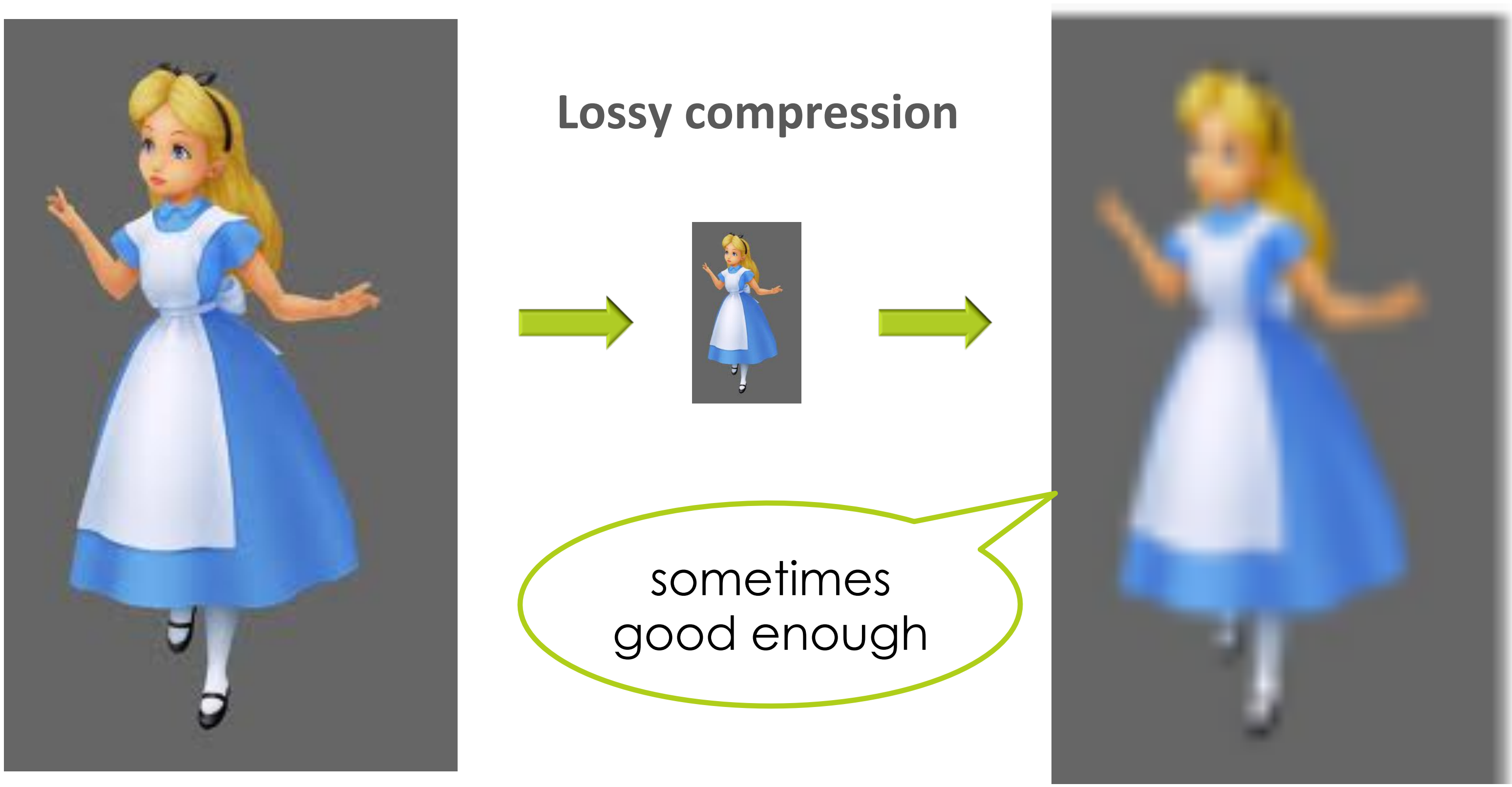
- **Lossless** –
  - encodes the original information exactly.
- **Lossy** –
  - approximates the original information.

Data Compression: choices

Lossless compression

good but
can be hard
to get

Data Compression: choices



Some Considerations

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

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

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?

(A)	(B)	(C)
023514	795	084664452746

Information

- More Digits = More Information
- Right?

Memorizing

Volunteer to memorize 10 digits

- 2737761413

Volunteer to memorize 100 digits

- 4444444444444444444444444444444444
  4444444444444444444444444444444444
  44444444444444444444444444444444

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

- 48599377668248052998391790815047514509135243 67800673622844553973169223820421306174607612 086978543115

Is that harder to memorize than:

- 44444444444444444444444444444444444444444444 44444444444444444444444444444444444444444444 444444444444

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

- In data compression our goal:
  - encode messages so that each bit conveys as much information as possible

Measuring information with: Algorithmic information theory

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

4859937766824805299839179081504751450913524367800673622844553973169223820421306174607612086978543115

```
print("48599377668248052998391790815047514509135243678006736228445539731692238204213061746076120869
78543115”)
```

Therefore,
Algorithmic Information Theory says:

48599377668248052998391790815047514
50913524367800673622844553973169223
820421306174607612086978543115

contains more information than

44444444444444444444444444444444444
44444444444444444444444444444444444
444444444444444444444444444444

Pi and information

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

Pi 10000

**3141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117067982148086513282306
6470938446095505822317253594081284811174502841027019385211055596446229489549303819644288109756659334461284756482337867
8316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789
2590360011330530548820466521384146951941511609433057270365759591953092186117381932611793105118548074462379962749567351
8857527248912279381830119491298336733624406566430860213949463952247371907021798609437027705392171762931767523846748184
6766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995
6112129021960864034418159813629774771309960518707211349999998372978049951059731732816096318595024459455346908302642522
3082533446850352619311881710100031378387528865875332083814206171776691473035982534904287554687311595628638823537875937
5195778185778053217122680661300192787661119590921642019893809525720106548586327886593615338182796823030195203530185296
8995773622599413891249721775283479131515574857242454150695950829533116861727855889075098381754637464939319255060400927
7016711390098488240128583616035637076601047101819429555961989467678374494482553797747268471040475346462080466842590694
9129331367702898915210475216205696602405803815019351125338243003558764024749647326391419927260426992279678235478163600
9341721641219924586315030286182974555706749838505494588586926995690927210797509302955321165344987202755960236480665499
1198818347977535663698074265425278625518184175746728909777727938000816470600161452491921732172147723501414419735685481
6136115735255213347574184946843852332390739414333454776241686251898356948556209921922218427255025425688767179049460165
3466804988627232791786085784383827967976681454100953883786360950680064225125205117392984896084128488626945604241965285
0222106611863067442786220391949450471237137869609563643719172874677646575739624138908658326459958133904780275900994657
6407895126946839835259570982582262052248940772671947826848260147699090264013639443745530506820349625245174939965143142
9809190659250937221696461515709858387410597885959772975498930161753928468138268683868942774155991855925245953959431049
9725246808459872736446958486538367362226260991246080512438843904512441365497627807977156914359977001296160894416948685
5584840635342207222582848864815845602850601684273945226746767889525213852254995466672782398645659611635488623057745649
8035593634568174324112515076069479451096596094025228879710893145669136867228748940560101503308617928680920874760917824
9385890097149096759852613655497818931297848216829989487226588048575640142704775551323796414515237462343645428584447952
6586782105114135473573952311342716610213596953623144295248493718711014576540359027993440374200731057853906219838744780
8478489683321445713868751943506430218453191048481005370614680674919278191197939952061419663428754440643745123718192179
9983910159195618146751426912397489409071864942319615679452080951465502252316038819301420937621378559566389377870830390
6979207734672218256259966150142150306803844773454920260541466592520149744285073251866600213243408819071048633173464965
1453905796268561005508106658796998163574736384052571459102897064140110971206280439039759515677157700420337869936007230
5587631763594218731251471205329281918261861258673215791984148488291644706095752706957220917567116722910981690915280173
5067127485832228718352093539657251210835791513698820914442100675103346711031412671113699086585163983150197016515116851
7143765761835155650884909989859982387345528331635507647918535893226185489632132933089857064204675259070915481416549859
4616371802709819943099244889575712828905923233260972997120844335732654893823911932597463667305836041428138830320382490
3758985243744170291327656180937734440307074692112019130203303801976211011004492932151608424448596376698389522868478312
3552658213144957685726243344189303968642624341077322697802807318915441101044682325271620105265227211166039666557309254
7110557853763466820653109896526918620564769312570586356620185581007293606598764861179104533488503461136576867532494416
6803962657978771855608455296541266540853061434443185867697514566140680070023787765913440171274947042056223053899456131
4071127000407854733269939081454664645880797270826683063432858785698305235808933065757406795457163775254202114955761581
4002501262285941302164715509792592309907965473761255176567513575178296664547791745011299614890304639947132962107340437
5189573596145890193897131117904297828564750320319869151402870808599048010941214722131794764777262241425485454033215718
5306142288137585043063321751829798662237172159160771669254748738986654949450114654062843366393790039769265672146385306
7360965712091807638327166416274888800786925602902284721040317211860820419000422966171196377921337575114959501566049631
8629472654736425230817703675159067350235072835405670403867435136222247715891504953098444893330963408780769325993978054
193414473774418426312986080998888**

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).

Measuring information with: Shannon Information Theory

Idea 2: Shannon information theory

- We measure information content in bits
  - We can represent 2^k different symbols with k bits.
  - Turn the idea around: if we want to represent M different things, we need $\log_2 M$ bits

- **But** this is only true if the *M* things all have the same probability

Therefore

M =
48599377668248052998391790815047514
50913524367800673622844553973169223
820421306174607612086978543115

vs

M=
44444444444444444444444444444444444
44444444444444444444444444444444444
444444444444444444444444444444

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

- In data compression our goal:
  - encode messages so that each bit conveys as much information as possible

Therefore

M =
48599377668248052998391790815047514
50913524367800673622844553973169223
820421306174607612086978543115

vs

M=
44444444444444444444444444444444444
44444444444444444444444444444444444
444444444444444444444444444444

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!!!*
  - *56739594662393456*
- **High probability** events have **low** information content.
  - *The sun rose in the east this morning. Meh*
  - *4444444444444444*
- Low probability events need more bits than high
  - *Low probability events contain more information than high probability events*

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

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.

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

- Why do we care about entropy?
  - Tells us the minimum number of bits we need to encode each symbol in message M
  - Compression!

Data Compression: Encoding

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 **m** is a sequence of characters in some alphabet A = {a_1, a_2, …, a_k}
  - e.g., A = the English alphabet,

Try 1: 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)**

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

- An example code table:

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

Try 1: Fixed length codes

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

`encode`(“**`deadbeef`**”)

011100000011001100100101

`d e a d b e e f`

What is **`decode`**(

“001000011010100011100”) ?

`b a d c e d e`

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

Codes

- A *codeword* is simply a binary string
- 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)

Try 2: A non-code example

- Code words don't all need to be the same length

x	code(x)
a	0
b	01
c	10

Try 2: 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

Try 3: Better, but more annoying…

- This code is fine in principle (everything is uniquely decodable).
- But decode is too hard. Try to decode

  00001011010011

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

Try 3: Better, but more annoying…

x	$\text{code}(x)$
a	00
b	01
c	001
d	011
e	11

- What is **`decode(`** "0000101101001 1"**`)`** ?

  `a c d b a e`

  - 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	$\text{code}(x)$
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	$\text{code}(x)$
a	0
b	11
c	10

Prefix codes are uniquely decodable

Let $\mathbf{b}_0\mathbf{b}_1\ldots\mathbf{b}_n$ be the bits of a coded message.

Read off the bits from left to right until
$\mathbf{b}_0\mathbf{b}_1\ldots\mathbf{b}_k$ = **`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.

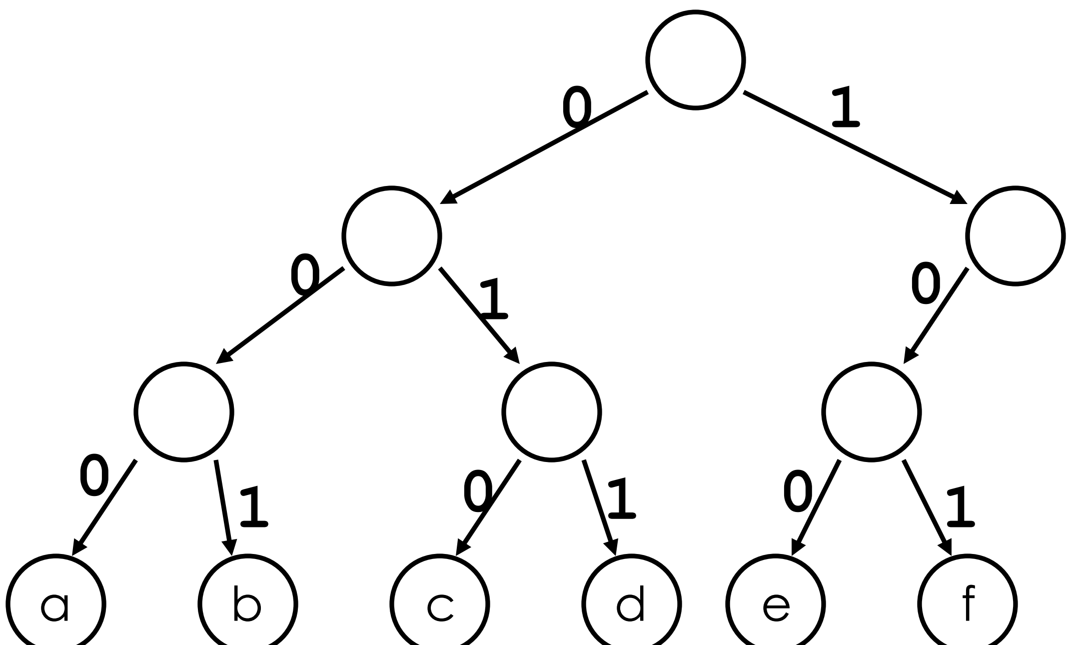
- Note: Prefix codes require no lookahead.

Data Compression: Decoding

Decoding prefix coded messages

Use a binary "prefix" tree

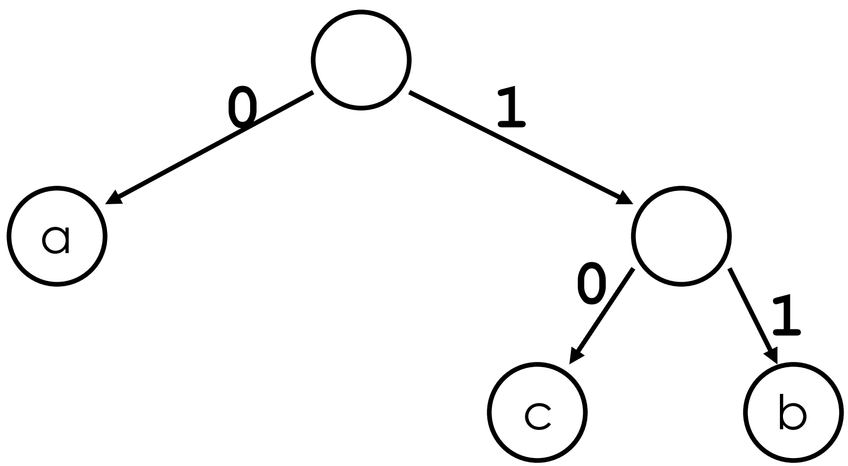
- Start at root, walk left for each "0", walk right for each "1" until you reach a leaf
- Return to root after you decode a character



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

Use a binary "prefix" tree

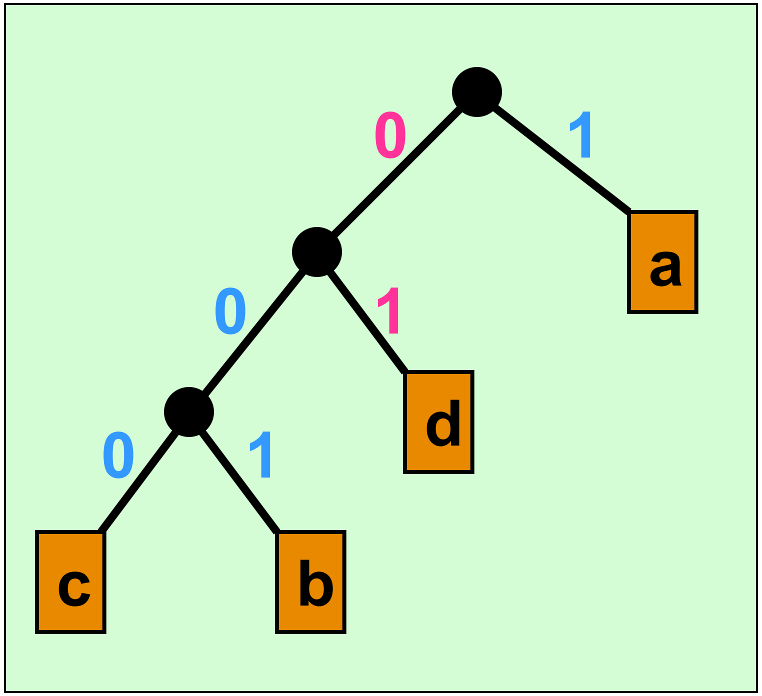
- Start at root, walk left for each "0", walk right for each "1" until you reach a leaf
- Return to root after you decode a character



x	code(x)
a	0
b	11
c	10

An optimal prefix tree is Full

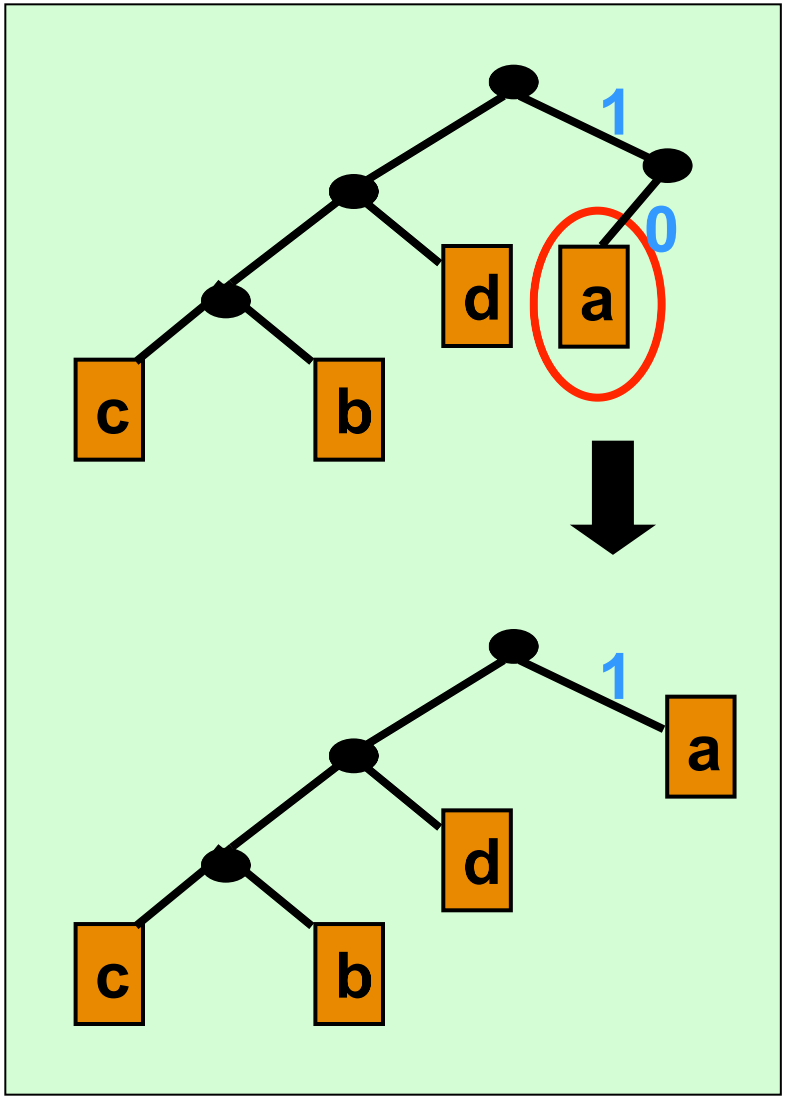
- *Full*: every node
  - Is a leaf, or
  - Has *exactly* 2 children.



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

Why a full binary tree?

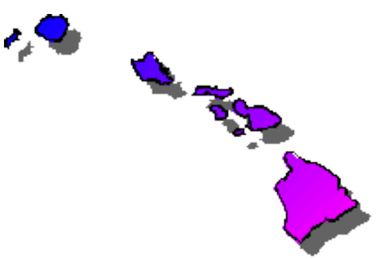
- A node with no sibling can be moved up 1 level, improving the code.
- An *optimal* prefix code for a string can *always* be represented by a full binary tree.



Huffman Codes

The Hawaiian Alphabet

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



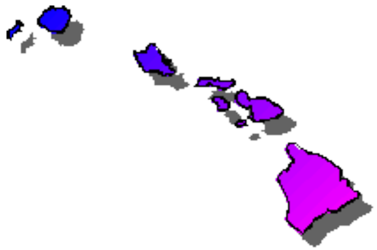
ʻ
A
E
H
I
K
L
M
N
O
P
U
W

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

The Hawaiian Alphabet: fixed-width encodings

- The Hawaiian alphabet consists of 13 characters.



```
'  →  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
```

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

Huffman Codes

- A type of optimal prefix code
- Commonly used for lossless data compression
- Developed by David A. Huffman
  - 1952, MIT
  - "A Method for the Construction of Minimum-Redundancy Codes"

Frequency counts as probabilities

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

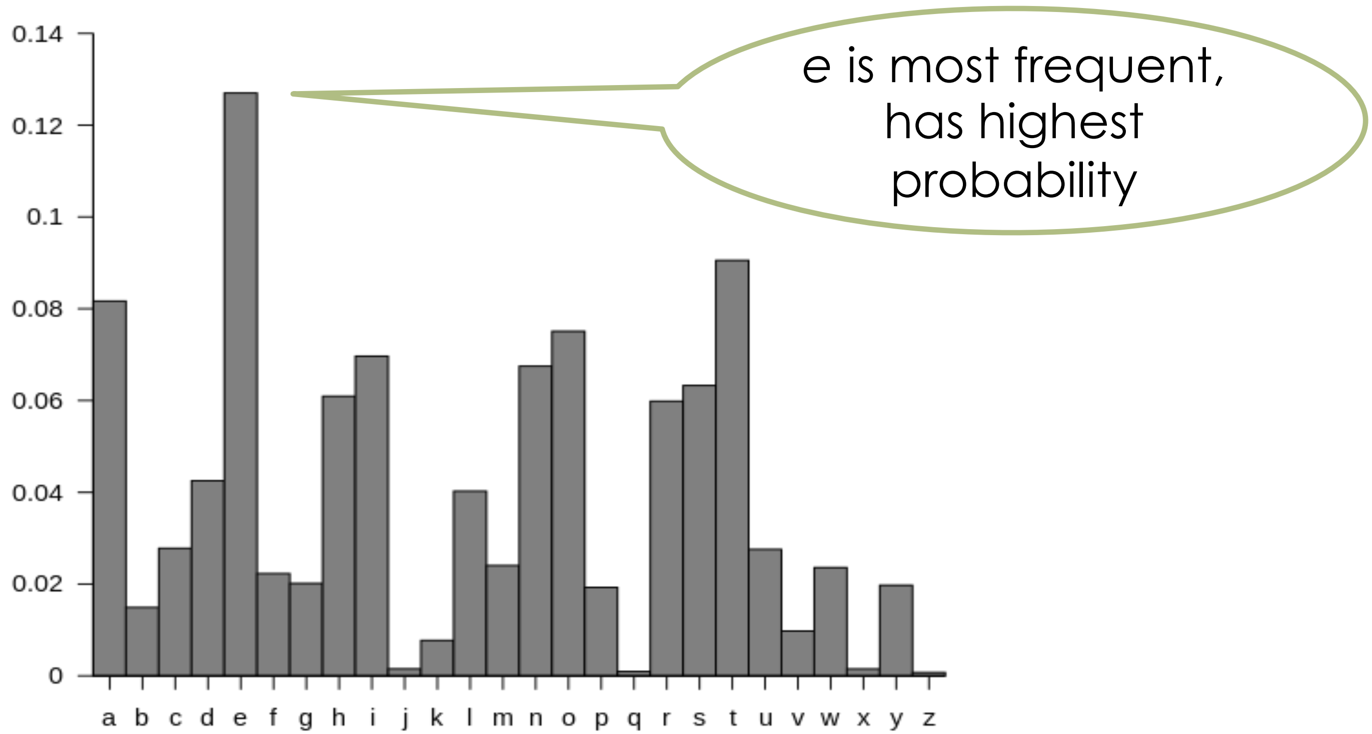


image: Wikipedia

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

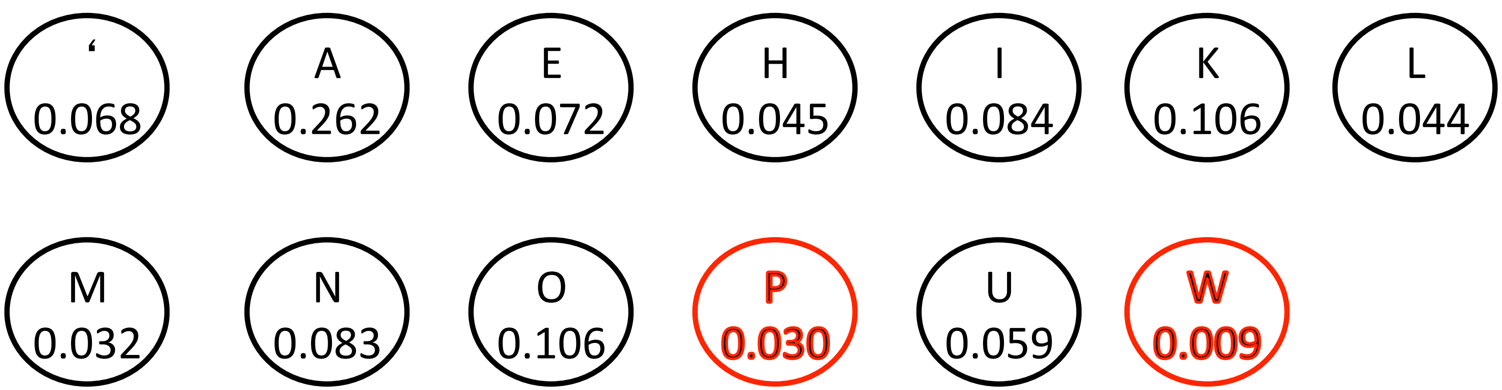
1. Assign character codes
    a. Obtain character frequencies
    b. Use frequencies to build a *Huffman tree*
    c. Use tree to assign variable-length codes to characters (store them in a table)
2. Use table to encode (compress) ASCII source file to variable-length codes
3. Use tree to decode (decompress)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

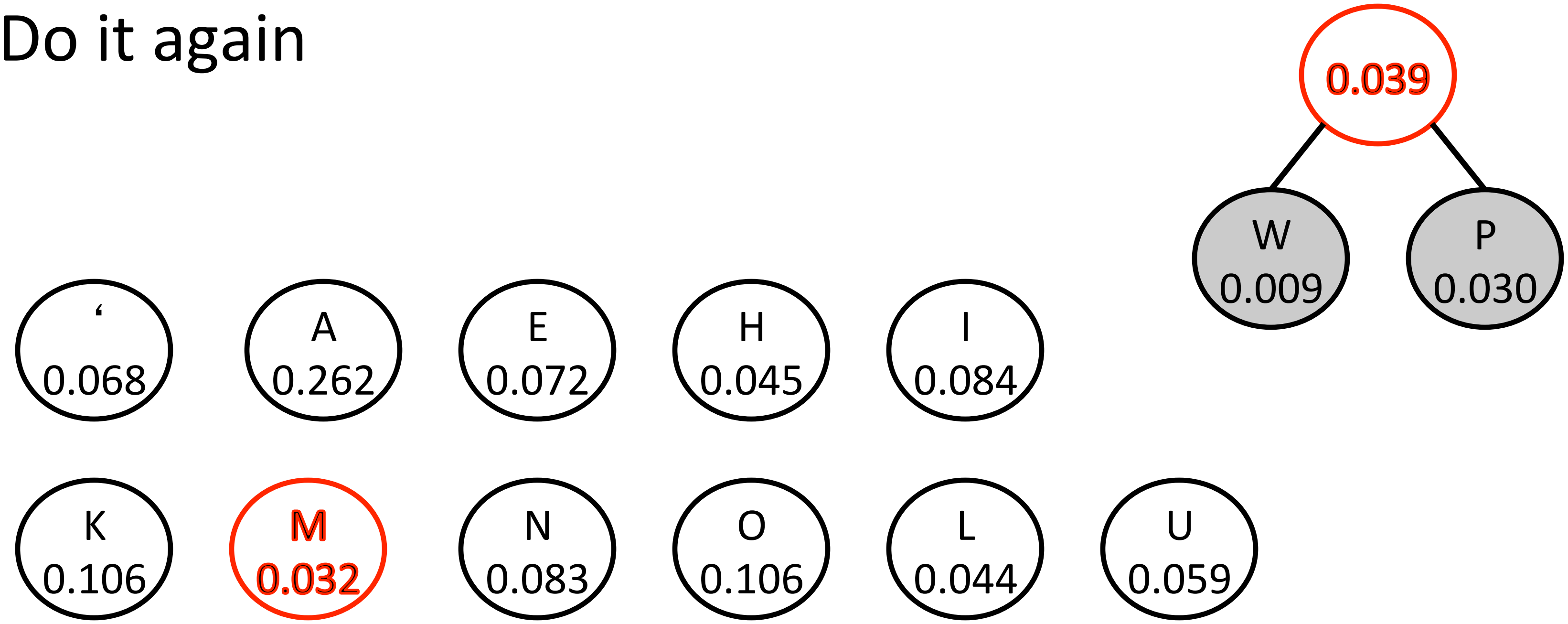
Building The Huffman Tree

- 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



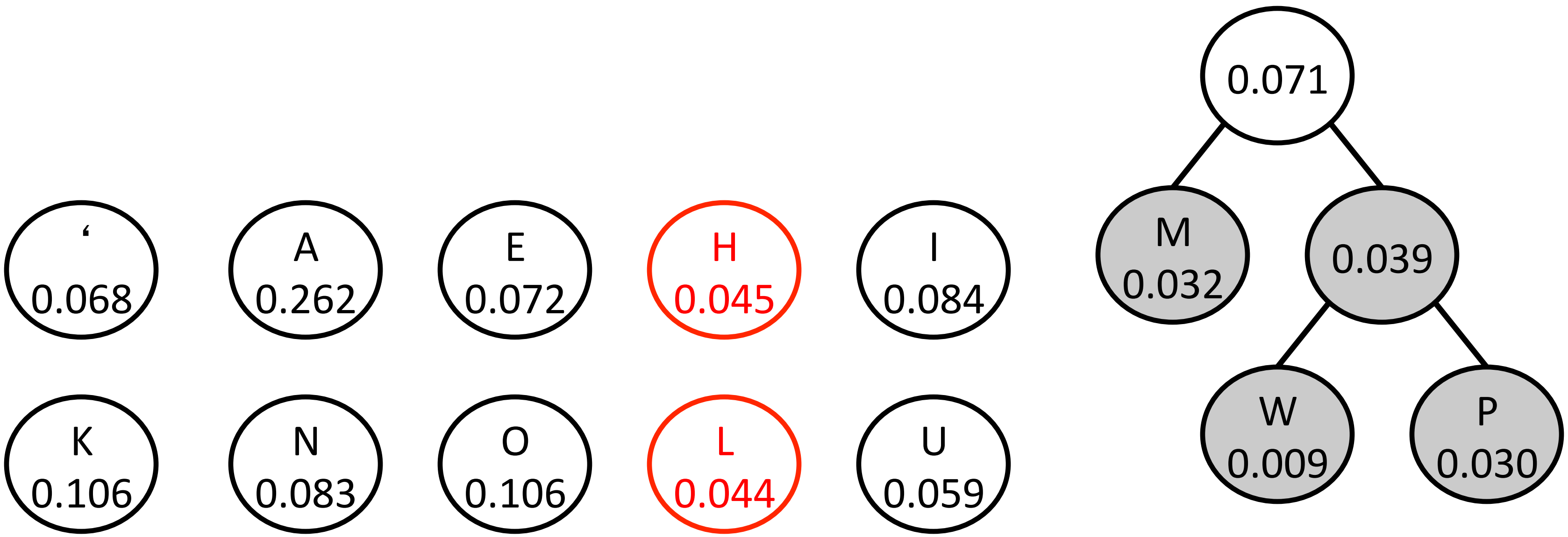
Building The Huffman Tree

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

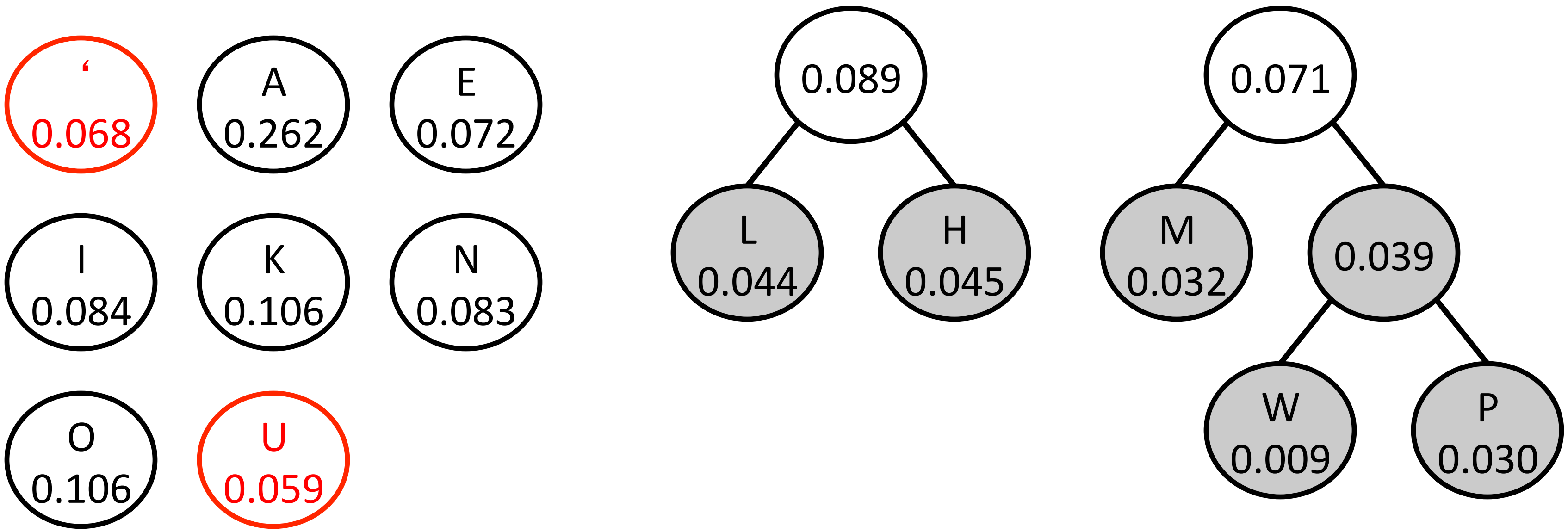


Building The Huffman Tree

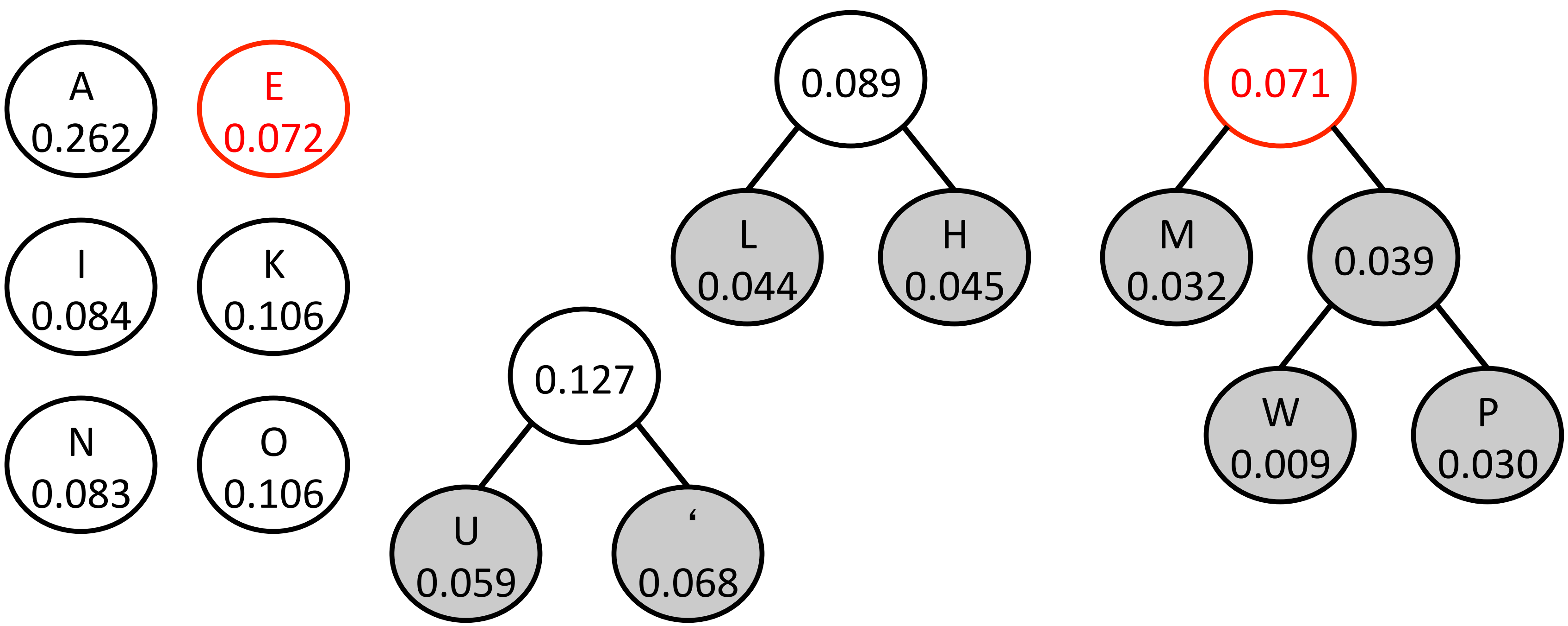
- …and again, as many times as possible



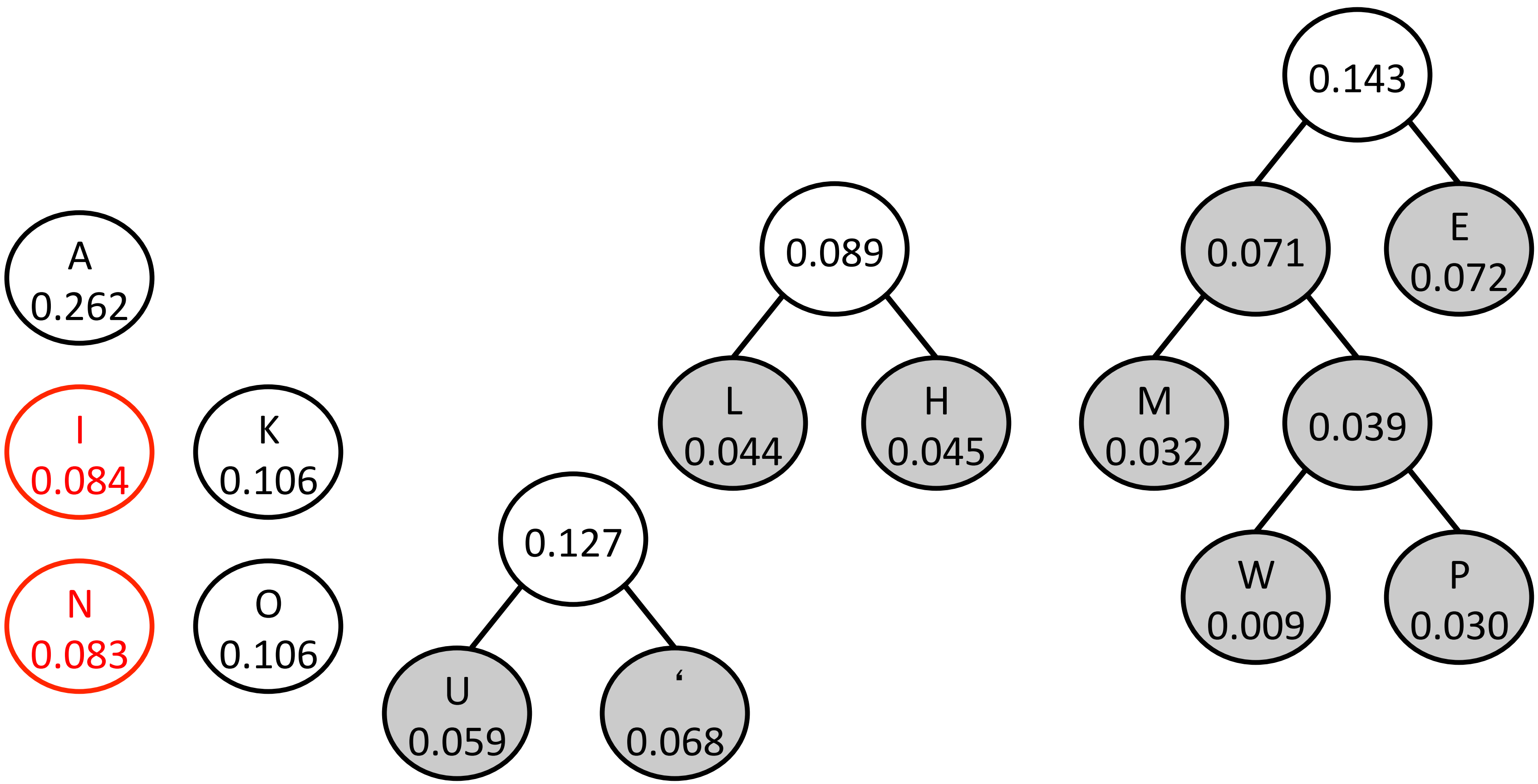
Building The Huffman Tree



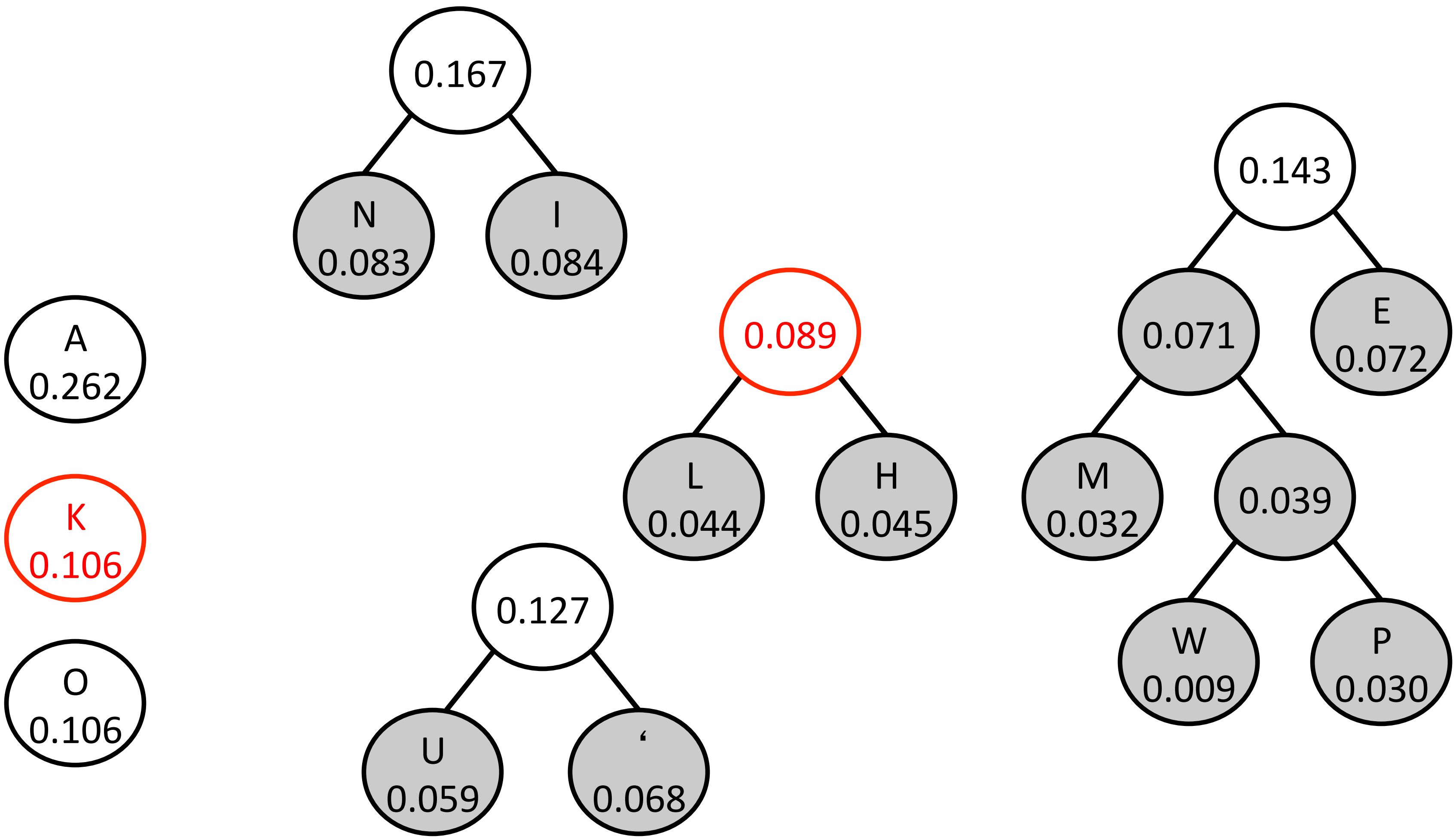
Building The Huffman Tree



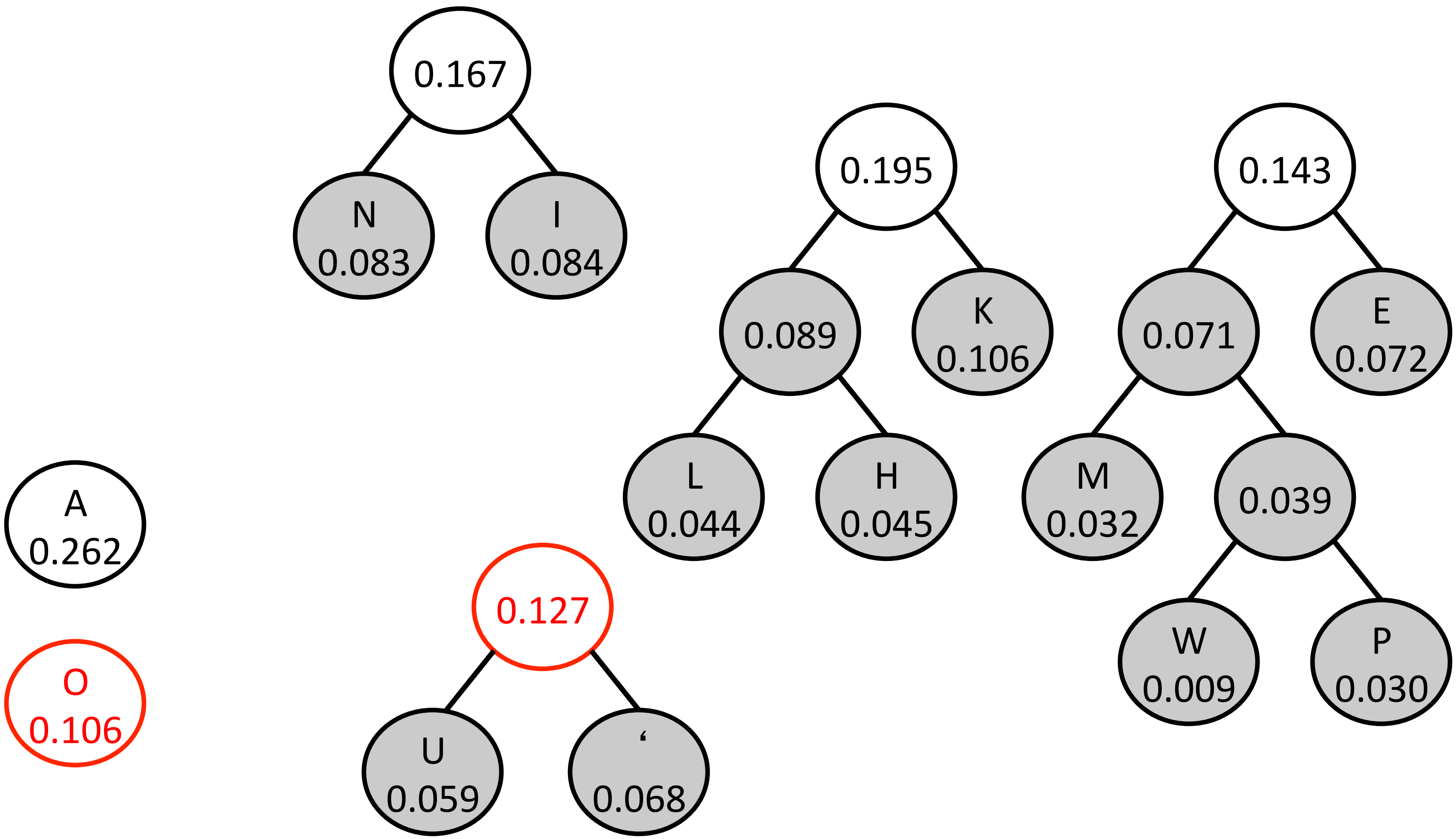
Building The Huffman Tree



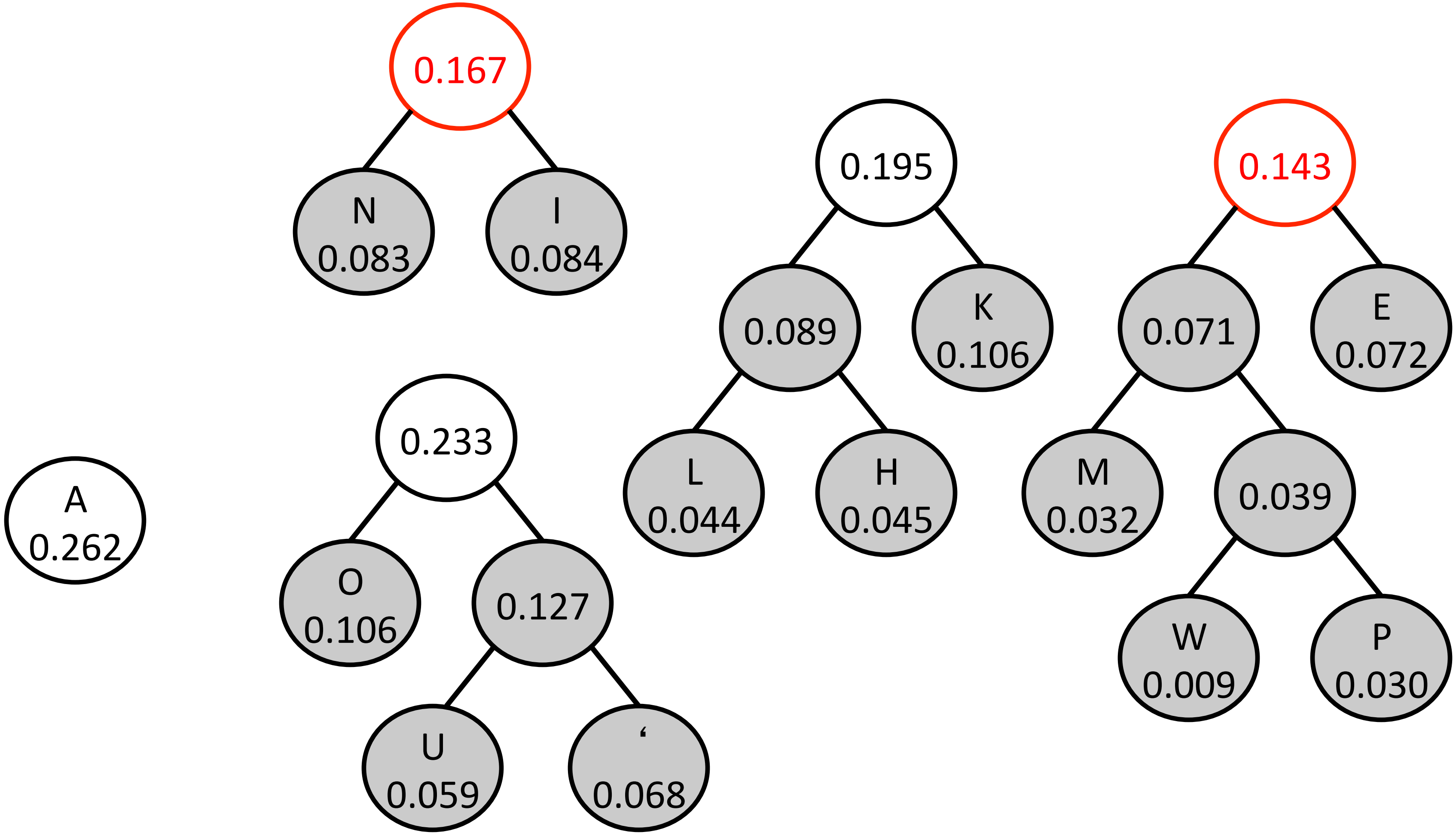
Building The Huffman Tree



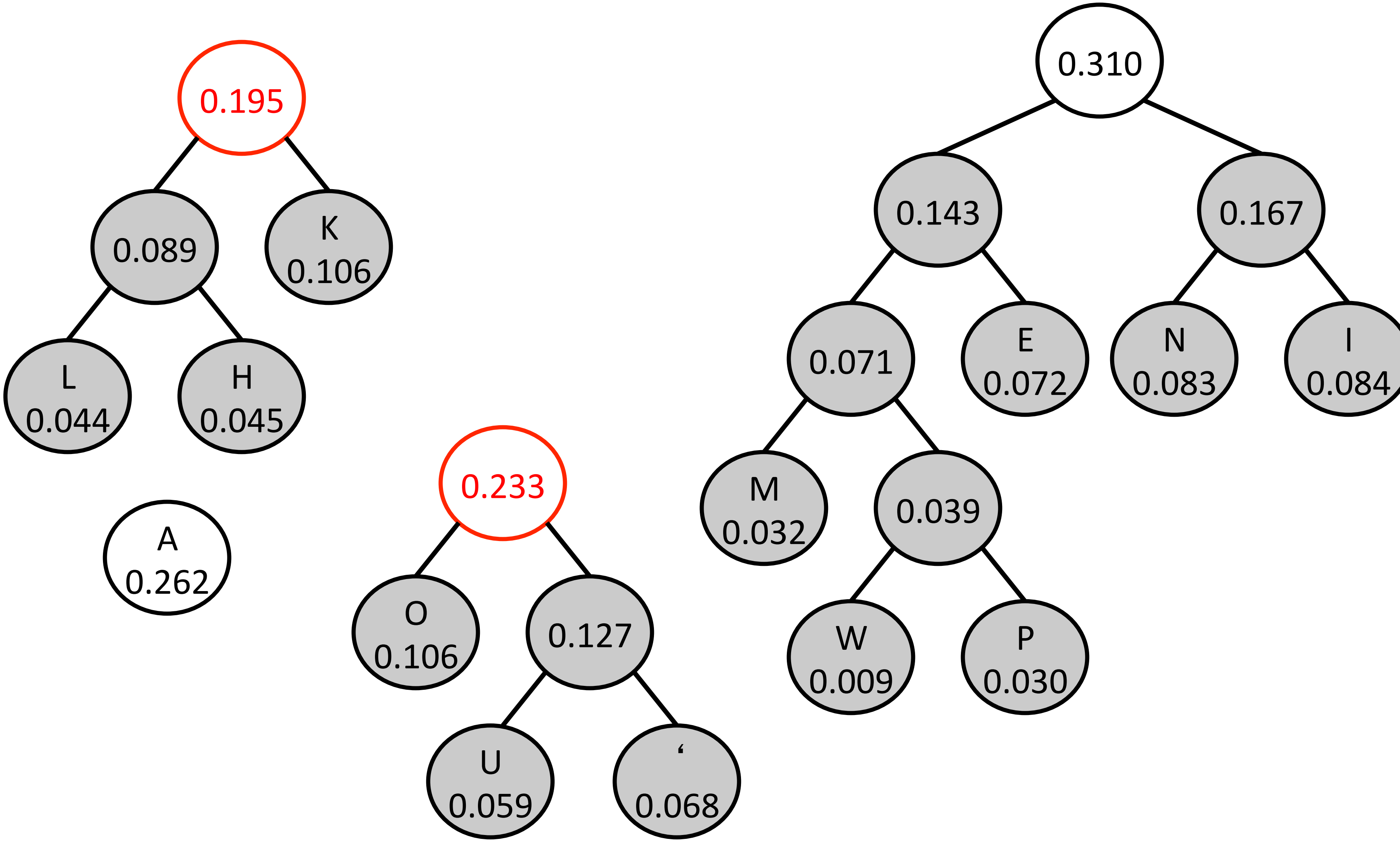
Building The Huffman Tree



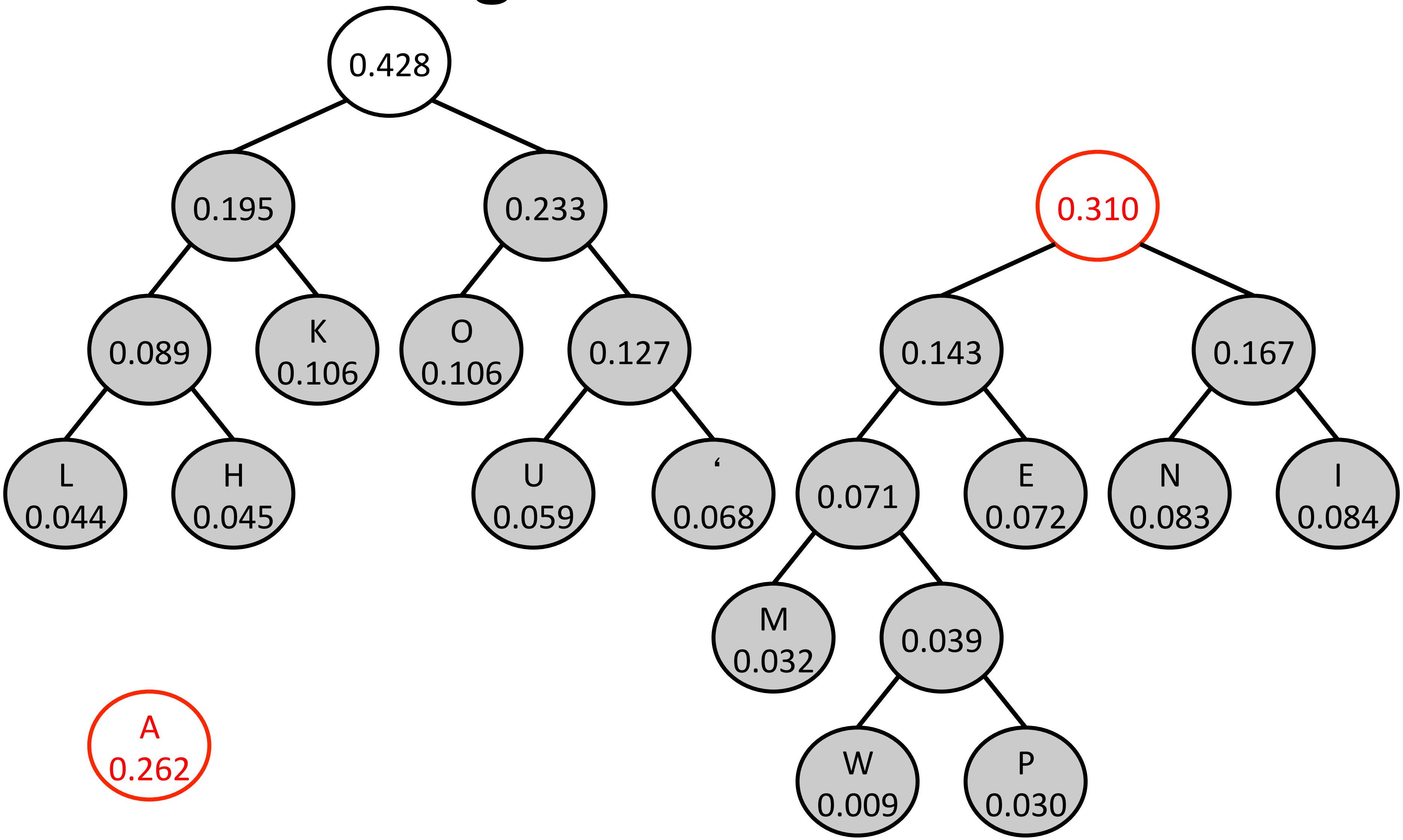
Building The Huffman Tree



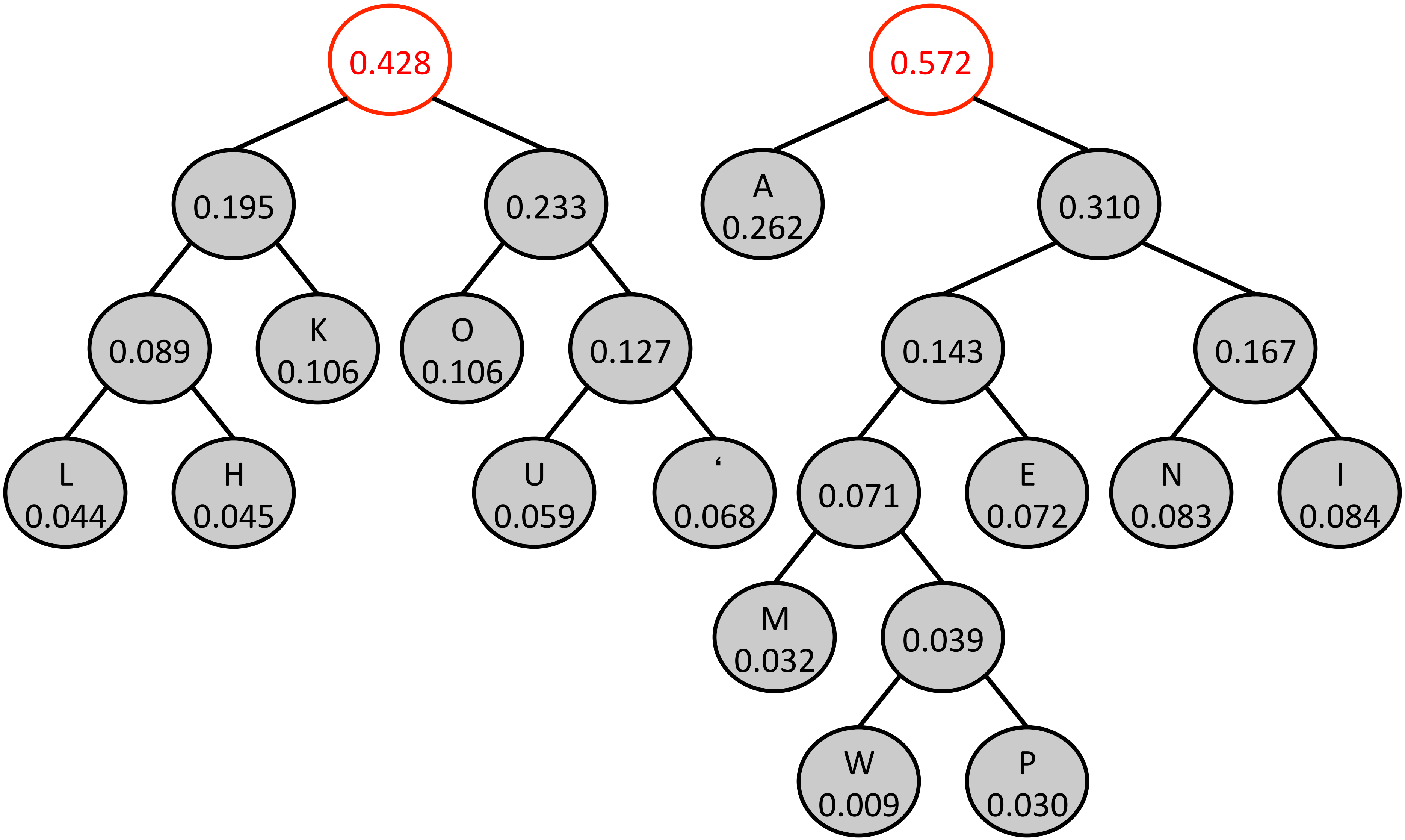
Building The Huffman Tree

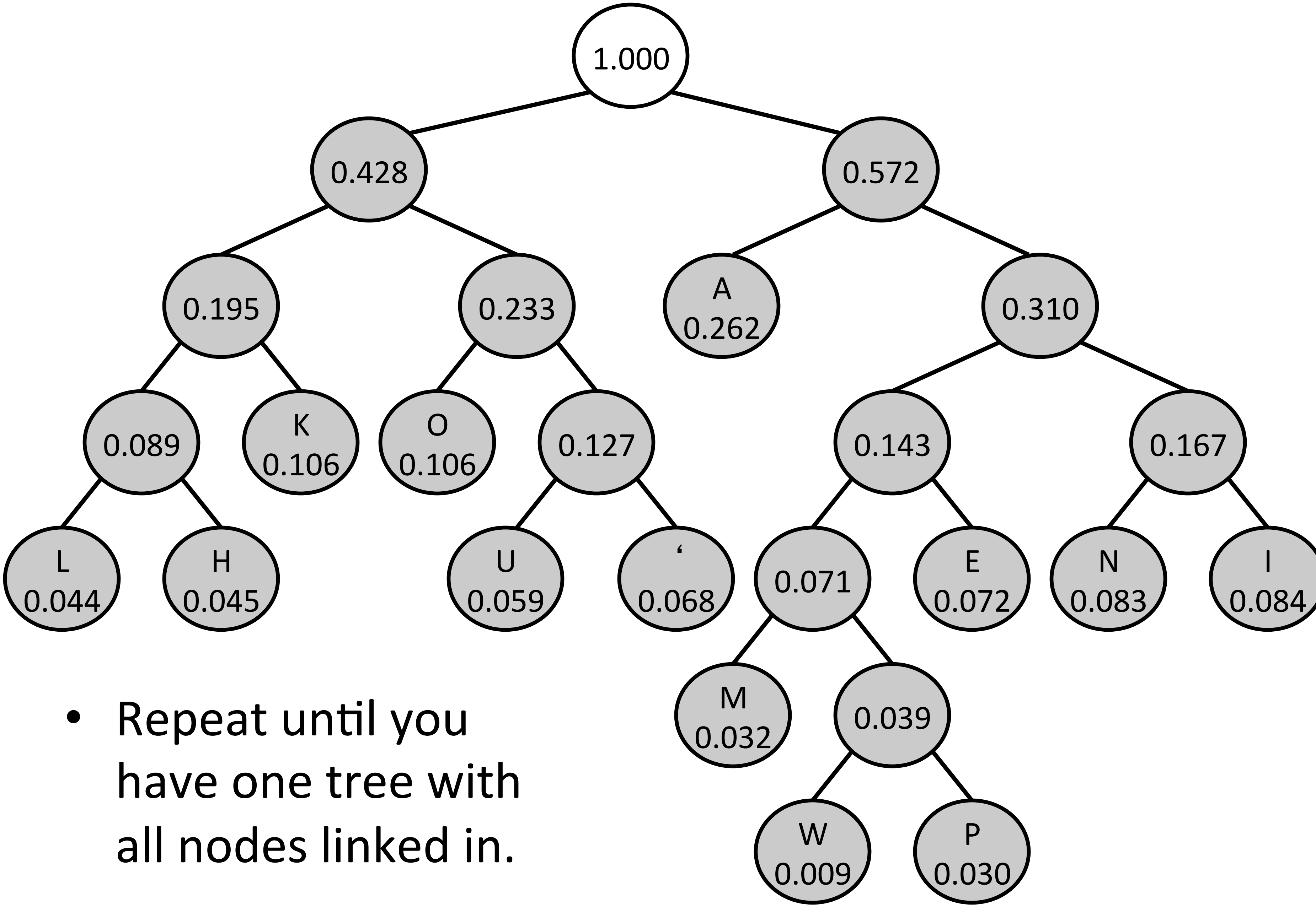


Building The Huffman Tree



Building The Huffman Tree

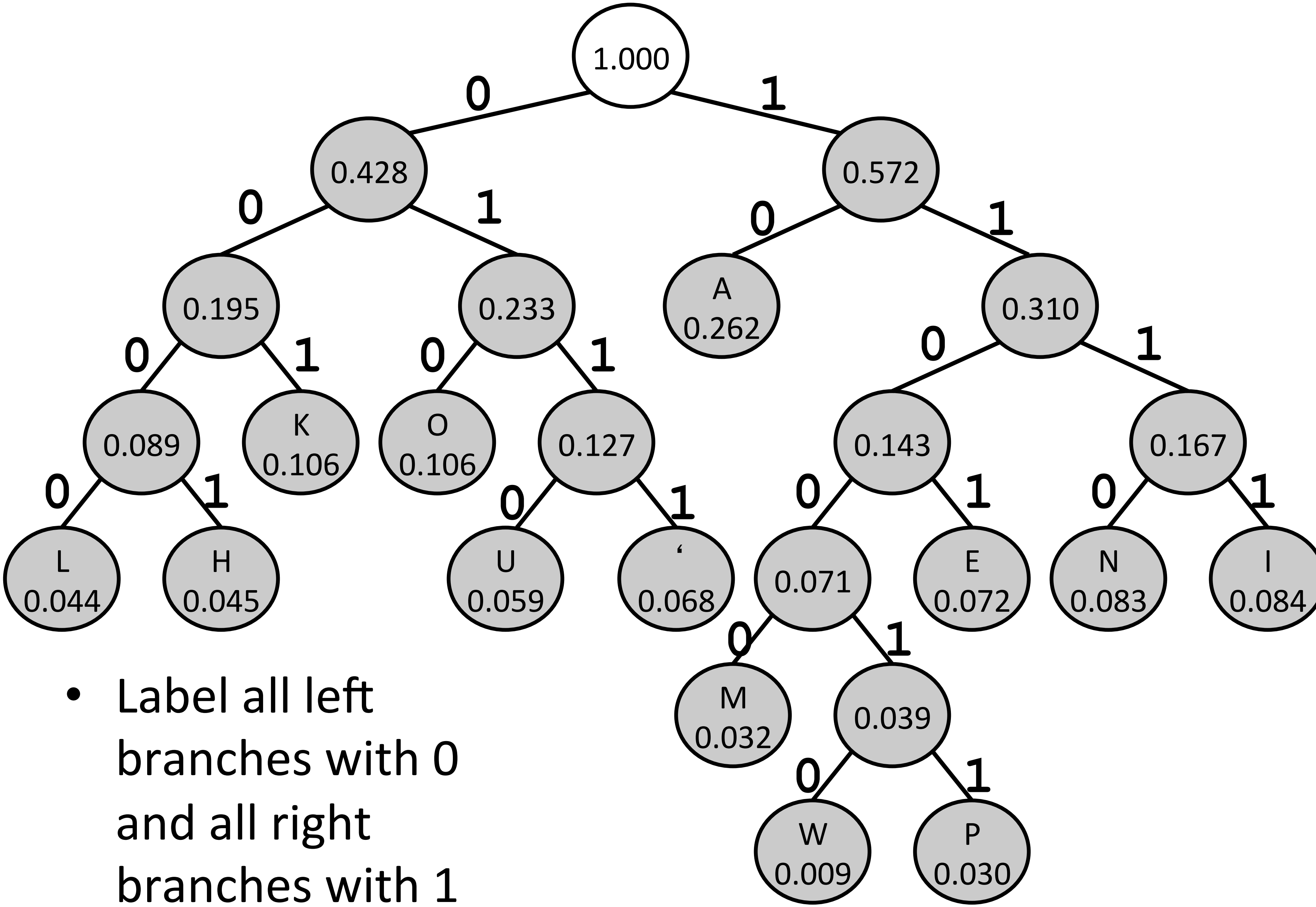




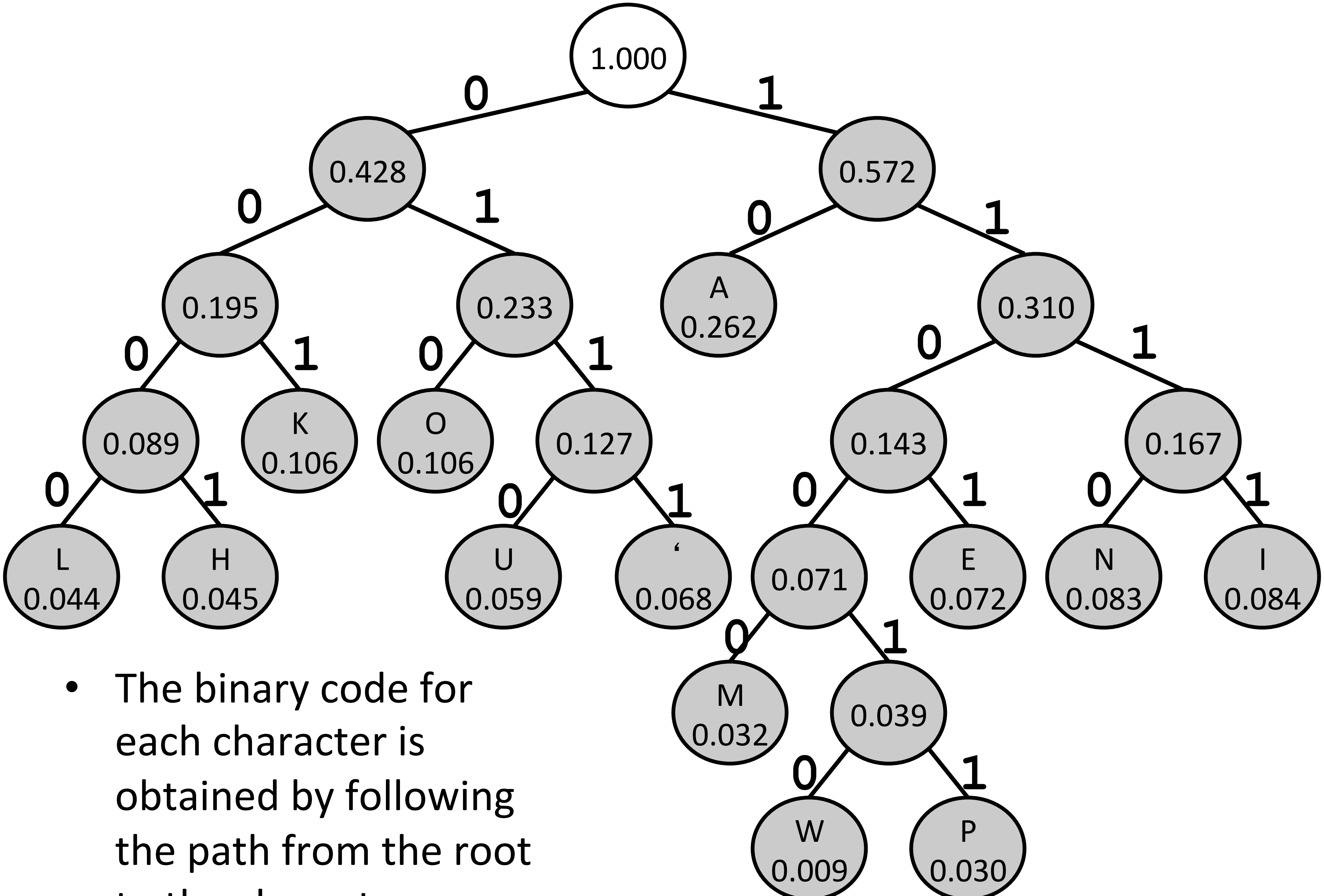
- Repeat until you have one tree with all nodes linked in.

Using the Tree to Assign Codes

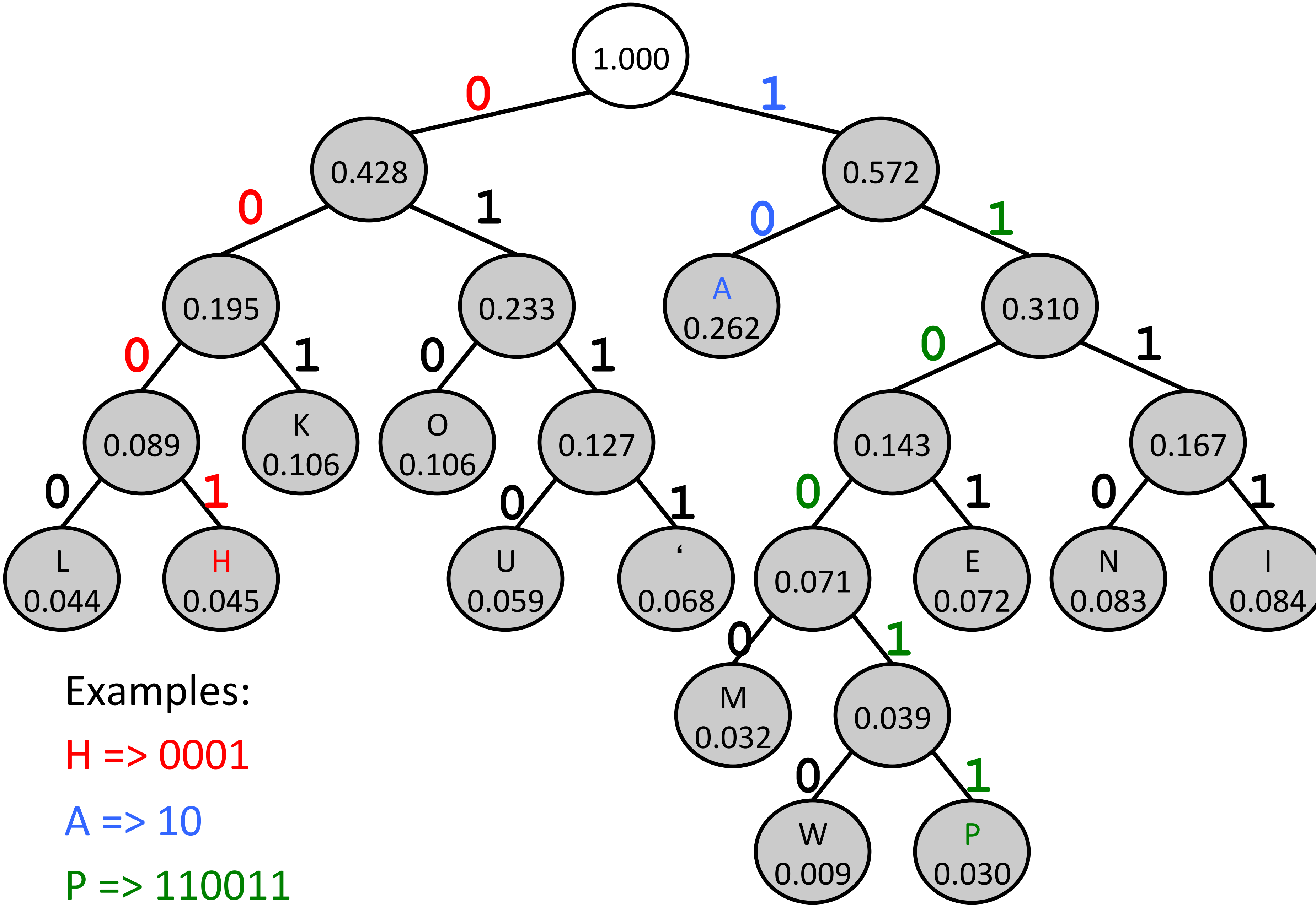
- The path from the root to each character determines the code



- 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)
  - Huffman code: 4454445444344434264242 = 84 bits
  - vs Fixed width encoding 22*4 = 88bits

How close did we get to minimum bits?

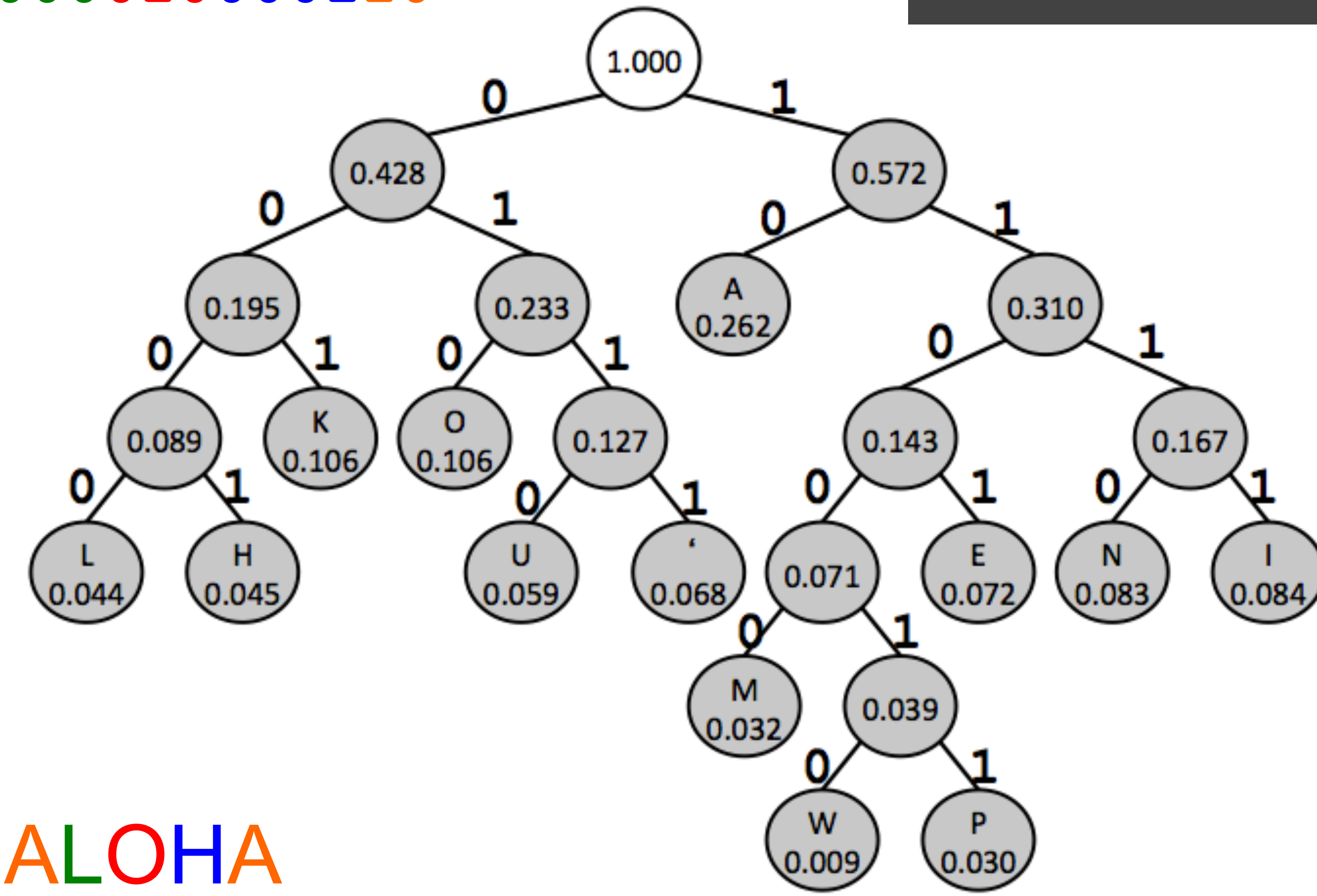
- We calculated the entropy as about 3.34 bits per character
- The average Huffman code length, weighted by the probabilities:

```
>>> 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	

Noisy network

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