Squeezing Ten Pounds of Data in a Five Pound Sack

- or -

The Compression Problem

Computational Thinking

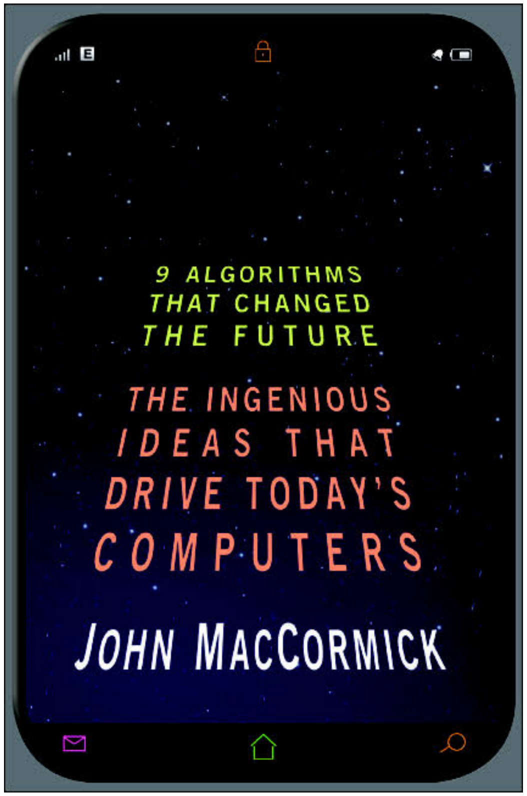
In this discussion, we will look at the compression problem.

Along the way, we will see several examples of computational thinking:

- **representations** allow the computer to store and work with letters, images, or music by representing them with a numeric code;
- **patterns** or repeated strings in text can be used by a computer for compression;
- **index tables** allow us to replace a long idea with a short word or number;

Reading assignment

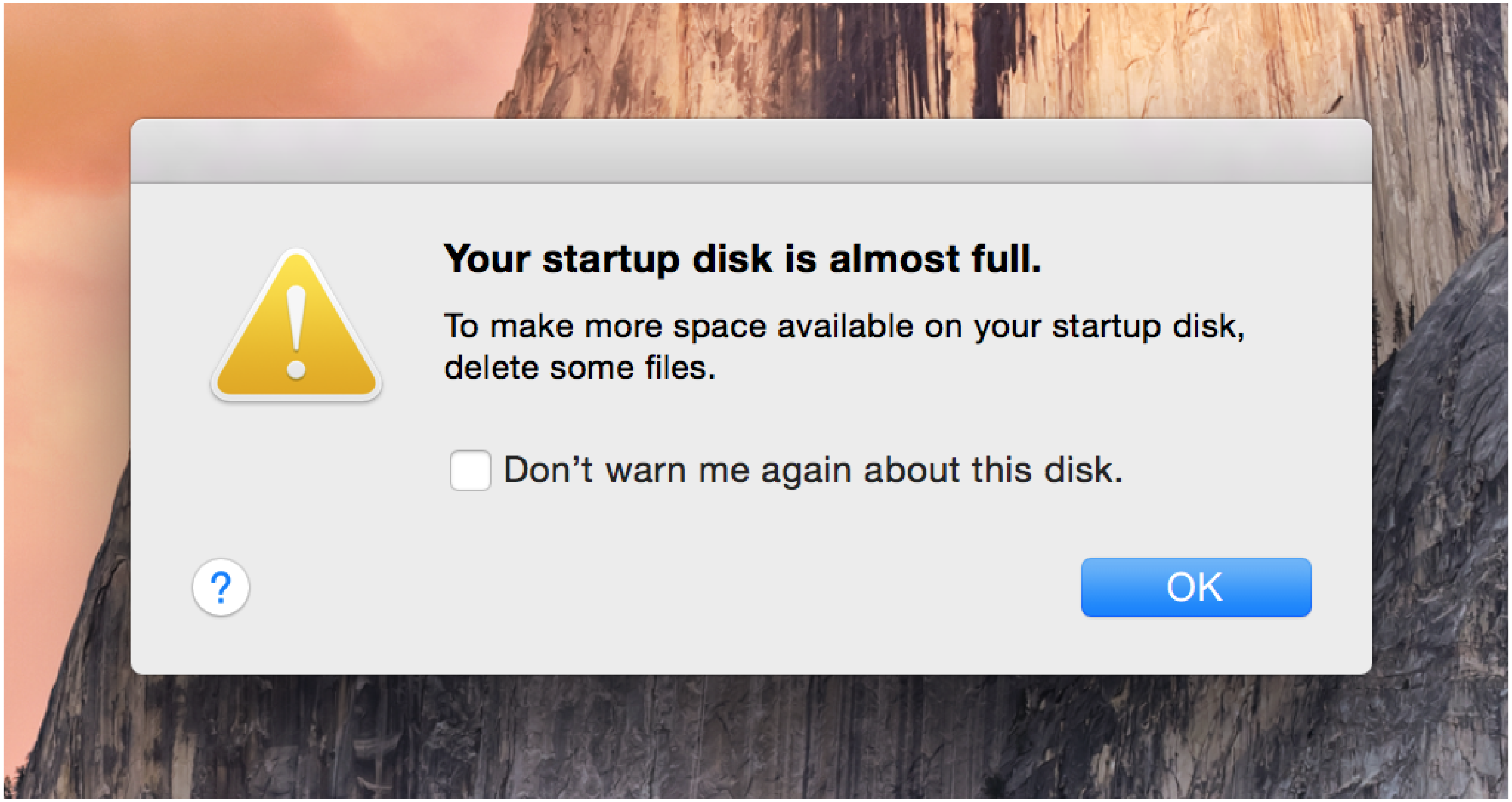
Read Chapter 7, pages 105–121, “9 Algorithms that Changed the Future”.



The Compression Problem for Traveling

When you pack for a trip, you want to minimize the number of suitcases and bags you take, but maximize the things you can transport. Sometimes it takes several repackings, rolling and squashing and rearranging, before you are satisfied.

The Compression Problem for File Storage



Your startup disk is almost full.

To make more space available on your startup disk, delete some files.

□ Don’t warn me again about this disk.

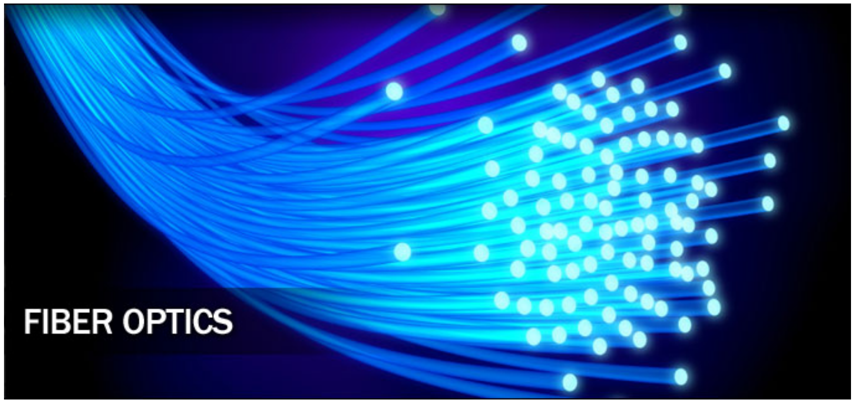
Your computer hard disk has a limited amount of storage. The information in a single movie or song could easily fill up your entire computer memory. This doesn’t happen because the information is compressed in several ways.

The Compression Problem for Data Transmission

Data must be transferred from one place to another, over networks with limited capacity. In North America, 70% of Internet traffic involves services that are streaming data, such as movies or music. When the local network is overloaded, a movie becomes unwatchable, music becomes noise, users are dissatisfied.

Solving the compression problem with more capacity

Dropbox



One solution to these problems is to increase capacity:

Buy more suitcases or bigger ones.

Buy another hard drive, or write stuff to a memory stick or DropBox.

Add more network servers, more cables, convert to fiber optics.

These temporary solutions all cost money and only help a few users.

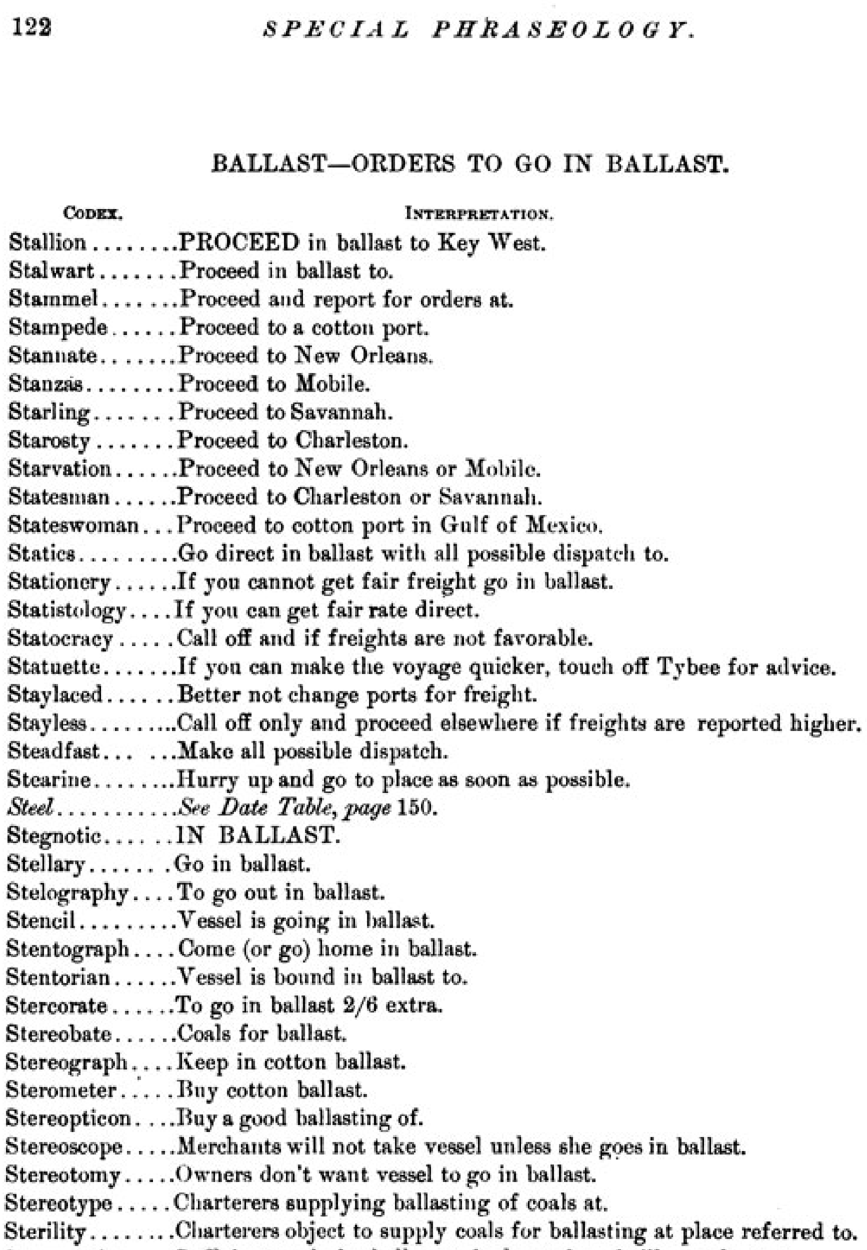
Solving the compression problem by shrinking things

Imagine if we could keep our suitcase, but shrink our clothes.

If the shrinking method is fast, safe, cheap, and reversible, then we have made a great improvement for everyone.

This is the idea behind data compression: don't buy bigger "buckets" and “pipes”, get smaller data instead!

Compression example: telegraph code



122 SPECIAL PHRASEOLOGY.

BALLAST—ORDERS TO GO IN BALLAST.

Codex.	Interpretation.
Stallion	PROCEED in ballast to Key West.
Stalwart	Proceed in ballast to.
Stammel	Proceed and report for orders at.
Stampede	Proceed to a cotton port.
Stannate	Proceed to New Orleans.
Stanzas	Proceed to Mobile.
Starling	Proceed to Savannah.
Starosty	Proceed to Charleston.
Starvation	Proceed to New Orleans or Mobile.
Statesman	Proceed to Charleston or Savannah.
Stateswoman	Proceed to cotton port in Gulf of Mexico.
Statics	Go direct in ballast with all possible dispatch to.
Stationery	If you cannot get fair freight go in ballast.
Statistology	If you can get fair rate direct.
Statocracy	Call off and if freights are not favorable.
Statuette	If you can make the voyage quicker, touch off Tybee for advice.
Staylaced	Better not change ports for freight.
Stayless	Call off only and proceed elsewhere if freights are reported higher.
Steadfast	Make all possible dispatch.
Stearine	Hurry up and go to place as soon as possible.
Steel	*See Date Table, page* 150.
Stegnotic	IN BALLAST.
Stellary	Go in ballast.
Stelography	To go out in ballast.
Stencil	Vessel is going in ballast.
Stentograph	Come (or go) home in ballast.
Stentorian	Vessel is bound in ballast to.
Stercorate	To go in ballast 2/6 extra.
Stereobate	Coals for ballast.
Stereograph	Keep in cotton ballast.
Sterometer	Buy cotton ballast.
Stereopticon	Buy a good ballasting of.
Stereoscope	Merchants will not take vessel unless she goes in ballast.
Stereotomy	Owners don't want vessel to go in ballast.
Stereotype	Charterers supplying ballasting of coals at.
Sterility	Charterers object to supply coals for ballasting at place referred to.

The telegraph code

Before the internet, radio, and telephone, people relied on the telegraph for communication. Sending a telegram could be time consuming and expensive. The user wrote the message on a standard form, went to a telegraph office, where an operator collected a fee based on the number of words, and then transmitted the message using short and long taps (dots and dashes) on a single key.

To save money, users shortened their messages. The message "Your mother is sick. Come home as soon as you can." would be written "MOM SICK. COME IMMEDIATELY."

Commercial users relied on sending hundreds of telegrams a day; the messages they sent were so simple that it was possible to write most of them down on a menu. They realized that instead of sending the message, they could just send the index of the message.

The telegraph code

The first idea was to give each message a numeric index. The message *If you cannot get fair freight, go in ballast* might be given numeric index 1747.

The numeric index idea had problems:

- A single mistake in the number would result in an entirely incorrect message.
- It's not easy to remember the number of a message;
- The telegraph company would get suspicious if numeric messages were sent;

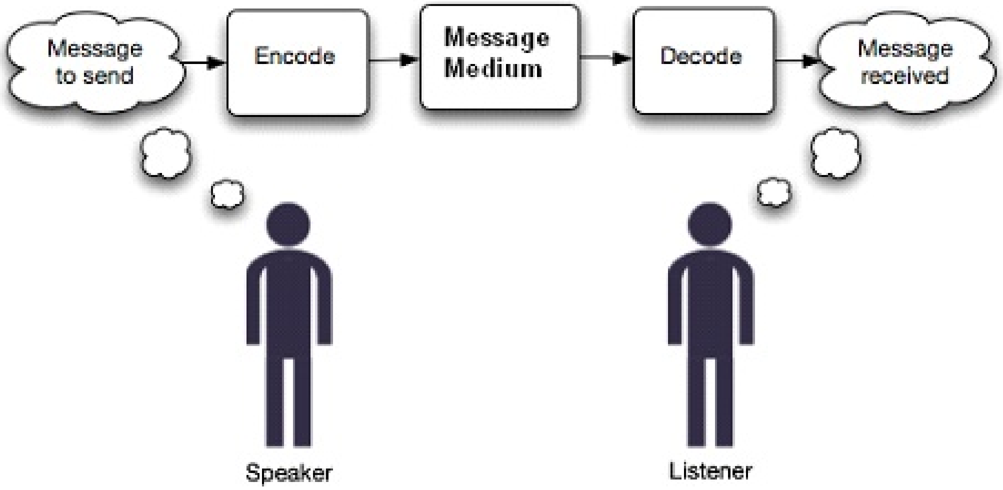
To avoid mistakes, it was decided to use index words, and not numbers, to encode the messages.

The message *If you cannot get fair freight, go in ballast* has an index word of **STATIONERY**;

- A mistake in the index word **STATIONARY** (oops!) is easy to spot.
- Words are easier to remember than numbers;
- The telegraph company won't complain about sending text message;

The telegraph code

So suppose the message *If you cannot get fair freight, go in ballast* is to be sent, using the index word **STATIONERY**; the recipient needs the code book in order to turn the one word telegram into the message.



The message goes through three steps, which we can think of as **ENCODE**, **TRANSMIT** and **DECODE**.

The encoding and decoding require access to the code book.

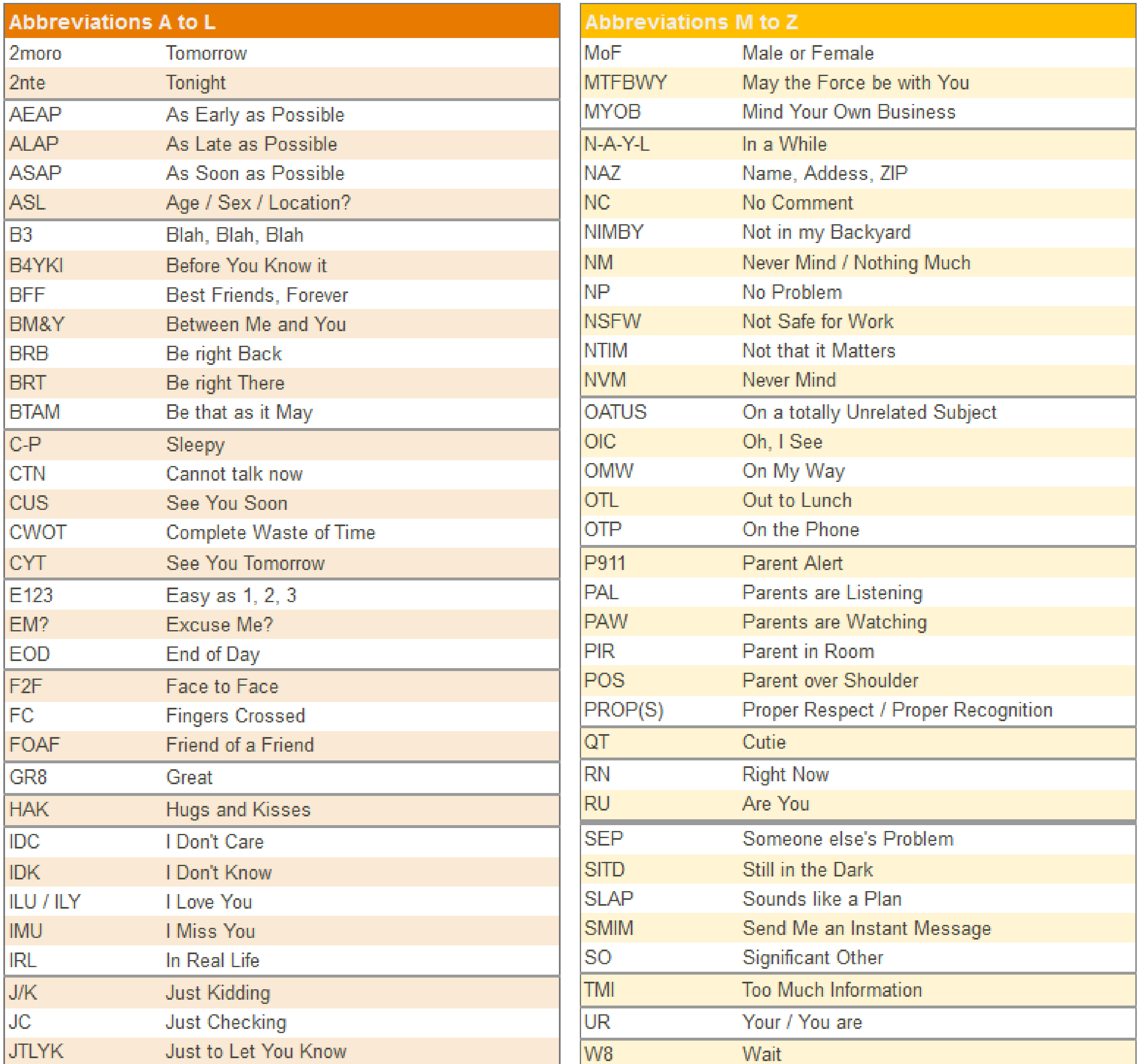
The telegraph code

Using the code book provides a measure of privacy; the telegraph clerk won't understand the message, and anyone seeing the message can't understand it without having access to the codebook. So, in a sense, the telegraph code is another example of a kind of **cryptography**.

However, the primary purpose of the telegraph code was not cryptography, but compression. By compressing a 9 word message to a single word, the sender cut down the cost of sending the message.

Reducing the size of a message is something we do all the time, when we use abbreviations, or references or slang in order to convey our meaning more quickly.

Compression example: Text messages



Abbreviations A to L	
2moro	Tomorrow
2nte	Tonight
AEAP	As Early as Possible
ALAP	As Late as Possible
ASAP	As Soon as Possible
ASL	Age / Sex / Location?
B3	Blah, Blah, Blah
B4YKI	Before You Know it
BFF	Best Friends, Forever
BM&Y	Between Me and You
BRB	Be right Back
BRT	Be right There
BTAM	Be that as it May
C-P	Sleepy
CTN	Cannot talk now
CUS	See You Soon
CWOT	Complete Waste of Time
CYT	See You Tomorrow
E123	Easy as 1, 2, 3
EM?	Excuse Me?
EOD	End of Day
F2F	Face to Face
FC	Fingers Crossed
FOAF	Friend of a Friend
GR8	Great
HAK	Hugs and Kisses
IDC	I Don't Care
IDK	I Don't Know
ILU / ILY	I Love You
IMU	I Miss You
IRL	In Real Life
J/K	Just Kidding
JC	Just Checking
JTLYK	Just to Let You Know

Abbreviations M to Z	
MoF	Male or Female
MTFBWY	May the Force be with You
MYOB	Mind Your Own Business
N-A-Y-L	In a While
NAZ	Name, Address, ZIP
NC	No Comment
NIMBY	Not in my Backyard
NM	Never Mind / Nothing Much
NP	No Problem
NSFW	Not Safe for Work
NTIM	Not that it Matters
NVM	Never Mind
OATUS	On a totally Unrelated Subject
OIC	Oh, I See
OMW	On My Way
OTL	Out to Lunch
OTP	On the Phone
P911	Parent Alert
PAL	Parents are Listening
PAW	Parents are Watching
PIR	Parent in Room
POS	Parent over Shoulder
PROP(S)	Proper Respect / Proper Recognition
QT	Cutie
RN	Right Now
RU	Are You
SEP	Someone else's Problem
SITD	Still in the Dark
SLAP	Sounds like a Plan
SMIM	Send Me an Instant Message
SO	Significant Other
TMI	Too Much Information
UR	Your / You are
W8	Wait

Text Messages

There are many shortcuts, abbreviations, and codes invented for texting.

Advantages include:

- **speed:** you're often texting while doing something else: sitting in class, or at lunch, or walking, and you don't have much time to get your message in.
- **space:** a cellphone doesn't have room to display a lengthy message, so short messages are preferred.
- **efficiency:** it's often awkward to type on the keypad of a cellphone, and so shortening the message reduces the work.
- **privacy:** if you use abbreviations, parents and other outsiders may be unable to decode your messages.

Text Messages

The abbreviations for text messages can about spontaneously, without any organization. Users just invented them as shortcuts, and other people liked them and started using them too. They work because:

- A lot of texting involves a limited number of messages of a standard form - I'm sad, I'm laughing, I agree, that's wonderful, I'm alone, that's just my opinion; this means it can be worth the effort to come up with a system for shortening the messages;
- It's easy to convert between the message and the abbreviations, since they are usually the first letters of the words in the phrase: ROFL = Rolling On the Floor Laughing;
- The short versions of the messages don't already mean anything; ROFL isn't already an English word, so seeing it, you know that it's a code for something else;

Aside from abbreviations, users leave out vowels and use other tricks:

If u cn rd this quickly, gd 4 u

Compression Example: Morse Code

Morse Code

The telegram was the earliest form of electronic communication.

A single wire extended from one city to another, and carried an electric current.

The telegram sender controlled a key which could interrupt or restore the current.

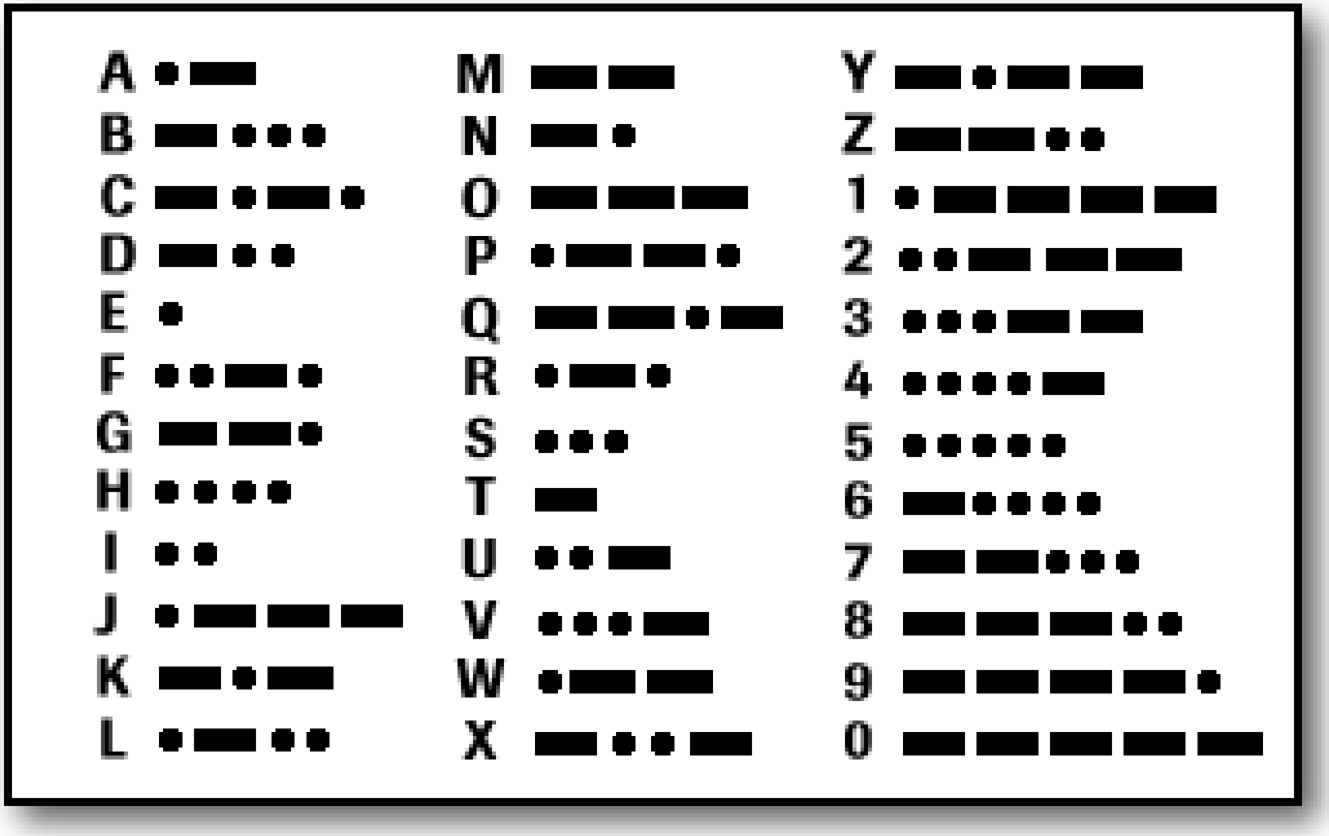
At the other end, the telegram receiver could hear the interruptions to the current.

By convention, senders used a combination of short and long interruptions to represent letters, with a longer pause between letters. The short and long interruptions were known as **dot** and **dash** respectively.

While a keyboard requires only a single stroke to type any letter, to transmit just the letters A through Z and digits 0 through 9 required anywhere from one to 5 strokes representing a mixture of dots and dashes.

Morse Code

Notice that we can represent A-Z and 0-9 using no more than 5 dots and dashes.



A	•—	M	——	Y	—•——
B	—•••	N	—•	Z	——••
C	—•—•	O	———	1	•————
D	—••	P	•——•	2	••———
E	•	Q	——•—	3	•••——
F	••—•	R	•—•	4	••••—
G	——•	S	•••	5	•••••
H	••••	T	—	6	—••••
I	••	U	••—	7	——•••
J	•———	V	•••—	8	———••
K	—•—	W	•——	9	————•
L	•—••	X	—••—	0	—————

Morse Code

There are two lessons here. The first is to be found in counting why we need up to five dots and dashes to represent 36 symbols.

Using exactly one symbol, we can represent two letters, as DOT and DASH.

If we use exactly two symbols, we can represent 4 letters, namely, DOT DOT, DOT DASH, DASH DOT and DASH DASH.

If we are allowed to use up to 2 symbols, we can represent 6 letters $(2 + 4)$.

Thinking this way, we can see that if we want to be able to represent the alphabet and digits (36 letters), the best way is to use combinations of DOT and DASH from 1 up to 5 in length.

Morse Code

As we have seen several times before, powers of 2 are important in figuring out the relationship between the number of symbols we can represent using a given number of DOT and DASH signals.

Number	Exactly	Up to
1	2	2
2	4	2+4=6
3	8	2+4+8=14
4	16	2+4+8+16=30
5	32	2+4+8+16+32=62

Morse Code

The second interesting fact involves the choice of which letters get the shortest Morse strings and which get the longest.

Let's group them by the length of their symbols. **E** and**T** come first, since they are DOT and DASH respectively:

Length	Number												
1	2	E	T										
2	4	A	I	M	N								
3	8	D	G	K	O	R	S	U	W				
4	16	B	C	F	H	J	L	P	Q	V	X	Y	Z
5	32	1	2	3	4	5	6	7	8	9	0		

Where have we seen something like this pattern before?

Remember **ETAIONSHRDLU** which we studied in cryptograms? The Morse code tries to set many of the common English letters to the shortest possible codes, to try to save time.

Compression Example: ZIP Files

Compression is a favorite tool of computers, servers and networks:

Messages sent over the Internet are compressed, transmitted, and then decompressed.

Audio information (telephone, CD, streaming music) is compressed; (moments of silence can be squeezed out, for starters, but much much more can be done.)

When you take a picture on your phone, and want to send it to someone, you are offered the choice of low, medium or high resolution versions of the picture to send.

Almost all software packages are downloaded in a compressed form and have to be decompressed before use.

The **ZIP** and **GZIP** formats are popular ways of compressing computer files to reduce the amount of storage required.

ZIP Files

As an example of how compression is used, let us consider the novel "The Life and Adventures of Robinson Crusoe" by Daniel Defoe. A text version of this book can be found at **http://www.gutenberg.org/ebooks/521**. The plain text version has size 634,873 bytes, which is the number of letters, spaces, punctuation marks and symbols.

```
The Life and Adventures of Robinson Crusoe,
by Daniel Defoe

CHAPTER I - START IN LIFE


I was born in the year 1632, in the city of York, of a good family,
though not of that country, my father being a foreigner of Bremen,
who settled first at Hull.  He got a good estate by merchandise, ...
```

ZIP Files

I can use the zip program to compress this file, as follows:

```
  zip robinson_crusoe.zip robinson_crusoe.txt
```

which creates a compressed version of size 227,924 bytes. Now that the file is compressed, it takes up much less space, but I can't actually read it. If you try to look at it, it seems like gibberish:

```
PK??,?0???f??D????1?$F0??1Y?0???S??.?7????/?~,0?^??<??....
```

where the question marks are actually weird characters that the computer can't really print out.

What happened to Robinson Crusoe? How did 634,873 bytes become 227,924 bytes? Did we throw things away? Is it all really still there somehow?

ZIP Files

Before we panic, let's check. If I want to read the novel, I just have to "unzip it", and the process of doing that might be as simple as saying:

```
  unzip robinson_crusoe.zip
```

Now my computer has the uncompressed file **robinson_crusoe.txt**, and it seems the same size, but to check that we can try to open it again:

```
The Life and Adventures of Robinson Crusoe,
by Daniel Defoe

CHAPTER I - START IN LIFE


I was born in the year 1632, in the city of York, of a good family,
though not of that country, my father being a foreigner of Bremen,
who settled first at Hull.  He got a good estate by merchandise, ...
```

ZIP Files

In fact, the ZIP program has a very interesting property.

ZIP was able to start with the original text of Robinson Crusoe and reduce it to about one third of its original size.

Then when we asked to UNZIP the file, it took the compressed version and expanded it.

What we got back was identical to the original file. Not a character or punctuation mark or symbol was different.

ZIP Files

Now the ZIP program is not magic. If it was, we could try to use ZIP twice in a row, and maybe squeeze the file down even further.

What actually happens if we try to compress the compressed file?

```
  zip twice.zip robinson_crusoe.zip
```

The compressed file has size 227,924 bytes. The twice compressed file has size 228,090 bytes, that is, it actually got slightly bigger. After you apply the ZIP program, the file has been squeeezed as much as ZIP can do, and it won't get any smaller.

Sometimes people have advertised compression programs that they say can be used repeatedly to make a file smaller each time the compression is applied. Such things are impossible! If you think about it, what happens when the file is compressed down to a single character and you want to compress it one more time? Does it vanish?

ZIP Files

Byte.com November 1995

In an announcement that sounded too good to be true, WEB Technologies (Smyrna, GA) claimed it had developed a compression algorithm that could squeeze almost any amount of data to less than 1024 bytes. The company claimed that its DataFiles/16 program could compress files larger than 64,000 bytes to about one-sixteenth their original size.

BYTE contacted WEB for a beta version of the software so that we could evaluate it. WEB at first declined to give us the beta, but we said we couldn't write a story about the product without one. WEB relented and sent us the beta version, which we tested and wrote about in the June 1992 issue.

Not surprisingly, the beta version of DataFiles/16 that reporter Russ Schnapp tested didn't work. DataFiles/16 compressed files, but when decompressed, those files bore no resemblance to their originals. WEB said it would send us a version of the program that worked, but we never received it.

When we attempted to follow up on the story about three months later, the company's phone had been disconnected.

Socrative Quiz PartVI_Practice_Quiz1

IUZGAZ34E

1. In Morse code, how many dots and dashes are needed to represent characters A-Z and 0-9? (Exactly 1, 1 or 2, no more than 5, 26)
2. In the telegraph code used by shippers, each message was replaced by a number. (True, False)
3. The abbreviations used in text messaging were simply invented by users and then became popular (True, False)
4. When the compressed text of "Robinson Crusoe" was uncompressed, not a single letter was different from the original text. (True, False)
5. If you run a compression program on a computer file over and over, it will keep getting smaller and smaller (True, False)

Two kinds of compression: lossy and lossless

There are many kinds of compression algorithms, but they fall into two types:

- lossless, uncompression brings back an exact copy of the original information;
- lossy, uncompression brings back only an approximate version of the original information; some is permanently lost.

Why would lossy compression ever be an option?

Given the choice between lossy and lossless compression, why would anyone choose the lossy option?

Lossy compression:

- can be much faster to do;
- can usually make much smaller compressed files;
- is useful on files that actually contain lots of redundant information;
- tends to discard information that is not really of much use.

Therefore, after first looking at lossless compression, we will want to come back to see how lossy compression works, and whether its advantages outweigh the permanent loss of some data.

Shipping code is lossless

Our shipping code is an example of a **lossless** compression method.

Suppose that the sender wants to send the message:
“If you cannot get fair freight, go in ballast.”

The codebook says to replace this by the single word **STATIONERY** which will be much cheaper to send.

When the message **STATIONERY** arrives, the receiver looks it up in the codebook and realizes that the intended (uncompressed) message is
“If you cannot get fair freight, go in ballast.”,
in other words, we recover exactly the original message, with no loss.

Shipping code is lossless, but one of the reasons it works is that it only expects to compress a certain fixed list of common statements in shipping.

How does lossless compression work?

The idea of lossless compression is simple: we start with a long message containing the information that we want to convey.

We notice that the information has patterns. By grouping or naming the patterns, we may shorten the message, while conveying the exact same information.

Consider, for example, the situation when someone when it would be possible to meet me for an hour next week.

I want to tell this person every hour that I am free, so that they can choose the time most convenient for them.

Assuming I work 8 hours in a day, 5 days a week, we have 40 hours to check.

My first draft of a message would then be 40 sentences, each one stating whether I am busy or free at a particular hour.

This certainly conveys the information, but we don't need a computer to realize that this message is much too long.

My busy 40 hour week

HOUR	Mon	Tue	Wed	Thu	Fri
9:00 -10:00	conference	prep			
10:00 -11:00	conference	meeting			
11:00 -12:00	class	meeting			
12:00 - 1:00	tutorial	meeting			
1:00 - 2:00	roundtable	roundtable		roundtable	roundtable
2:00 - 3:00	roundtable	roundtable		roundtable	roundtable
3:00 - 4:00	lecture	lecture			
4:00 - 5:00	lecture	lecture			

1: I am busy Monday at 9.
2: I am busy Monday at 10.
3: I am busy Monday at 11
...*(and on and on)*
40: I am free Friday at 4.

How does lossless compression work?

By looking at my schedule, I can see patterns. There are two days when I am very busy, and two other days when I am only a little busy.

Using patterns, I could compress this: *“Monday and Tuesday are full, and I’m booked from 1 to 3 on Thursday and Friday, otherwise I’m free.”*

The person receiving this message can perfectly reconstruct my calendar, that is, they can do a lossless decompression of my information.

This was only possible because my calendar actually did have patterns. Patterns don’t always occur, and if they’re not there, or we don’t spot them, then this simple idea is not much use.

Consider how much more difficult it would be to “compress” my schedule if it looked like the following one.

My chaotic 40 hour week

HOUR	Mon	Tue	Wed	Thu	Fri
9:00 -10:00	conference	prep		roundtable	
10:00 -11:00	conference		class		meeting
11:00 -12:00			lecture	lecture	
12:00 - 1:00	tutorial	meeting			
1:00 - 2:00		roundtable			roundtable
2:00 - 3:00	roundtable		lecture	roundtable	roundtable
3:00 - 4:00	roundtable			meeting	
4:00 - 5:00		lecture		roundtable	

Umm, I guess I'd better just list every hour!
I'm busy Monday at 9.
I'm busy Monday at 10.
I'm not busy Monday at 11.
I'm busy Monday at 12...*and on and on*

Run Length Encoding: A lossless compression method

When there are patterns in a message, a useful lossless compression method is known as run length encoding, or **RLE**.

Suppose I'm on the phone, reporting the 56 letter serial number on my computer to an operator, which is:

AAAAAAAAAAAAAAAAAAAAAAAABCBCBCBCBCBCBCBCBCBCAAAAAADEFDEFDEF

Rather than say each letter over the phone, I would surely describe it as:

21 A's, then 10 BC's, then 6 A's, then 3 DEF's

and if you wrote the message, you might write:

$$21A,10BC,6A,3DEF.$$

A message of 56 letters has been compressed to 16 letters, numbers and commas.

The compression ratio is the size of the original message divided by the size of the compressed message. In this case, our compression ratio is 56 / 16 = 3.5.

Computer data is full of runs, so RLE can help

Run length encoding is useful when messages or data often consist of letters or sets of letters that are immediately repeated several times.

Messages written in English don't have this property, but computer data is full of such situations.

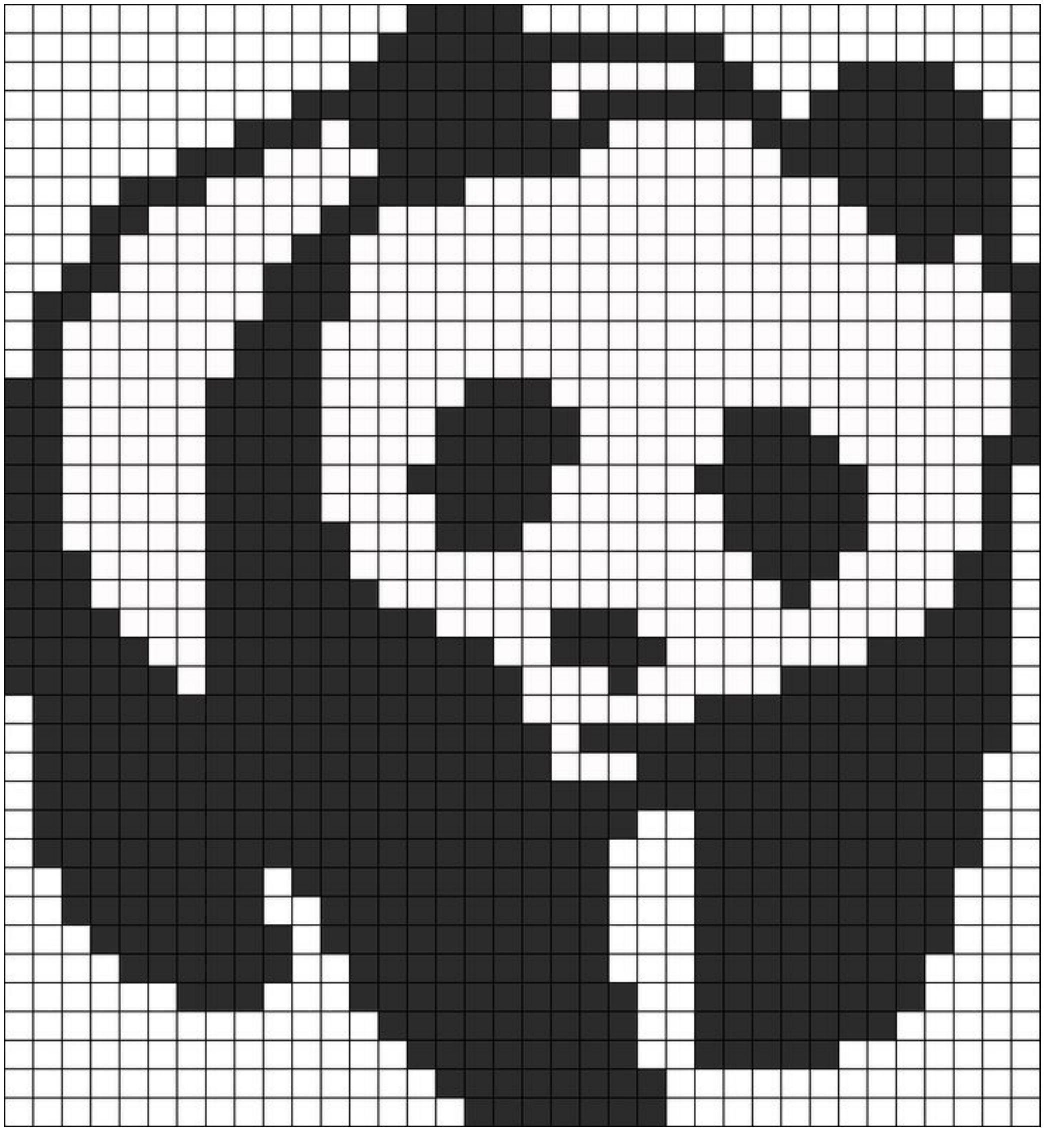
For example, cell phones and computers and televisions and digital cameras have to work with pictures.

To do this, they start with a representation of a picture, which is simply what looks like a grid, or piece of graph paper, consisting of square boxes, called pixels.

An image is defined by saying how many rows and columns of pixels are to be used, and what color is to be displayed inside each pixel.

A black and white image

The simplest computer image is a true black and white picture.



Describing a black and white image

To describe the panda image (39 columns, 36 rows), a computer could say:
White, White, White, ... or *W, W, W, ...*
listing the colors of each of 1,404 pixels, but that's ridiculously inefficient.

We can see that, if we "read" the image line by line, there are long sequences of white or black that can be described very simply.

Instead of describing the first row as 36 white or black values, we can simply say:
14 W, 4 B, 18 W,
that is, we are using run length encoding.

A simple run length encoding simply looks for stretches of all black or all white pixels, and such a method would not be able to compress a string like this:
W B W B W B W B W B W W B W B W B
but we can see that the appropriate shorter description is:
5 WB, 1 W, 3 WB
and smart RLE methods catch these patterns and lots of other ones as well.

Black/White images from faxes or scanners

An example where black/white pixel images are created is a fax machine, or the scanner that's available on some copying machines. These devices accept a document, take a picture of it, convert that picture to a pixel representation that can be transmitted over a computer network.

The fax or scanner "picture" is a grid of black or white dot squares created by sampling the document at regularly spaced locations, to create a representation of the document as an arrangement of pixels. This means that, whether the original document is text or a photograph, the fax message will consist of a description of how many rows and columns are used, followed by a string of 1's for black, and 0's for white.

A black/white image created on a scanner

```
@@.....@@.....@@@@@@@@@.....@@............@@@@@@@@.
@@.....@@.....@@............@@............@@.....@@
@@@@@@@@@.....@@@@@@@@@.....@@............@@@@@@@@.
@@.....@@.....@@............@@............@@.......
@@.....@@.....@@@@@@@@@.....@@@@@@@@@.....@@.......
```

This picture suggests how a scanner might “see” the word **HELP** on a document and create a 5 row x 51 column black and white image. using 255 characters. Here we use “@” for black and “.” for white.

RLE, applied to each row, reduces this to 147 characters:

```
2@,5.,2@,5.,9@,5.,2@,12.,8@,1.
2@,5.,2@,5.,2@,12.,2@,12.,2@,5.,2@
9@,5.,9@,5.,2@,12.,8@,1.
2@,5.,2@,5.,2@,12.,2@,12.2@,7.
2@,5.,2@,5.,9@,5.,9@,5.,2@,7.
```

for a compression ratio of $255/147 = 1.73$.

RLE misses a lot of compression opportunities

Run length encoding is a simple idea, and it misses many opportunities for compression because it only works if the repetitions are adjacent.

For instance, RLE can compress $\mathbf{ABABAB}$ but it can do nothing with the repeated $\mathbf{AB}$'s in the string $\mathbf{ABXABYABZ}$.

Because compression can be so useful, many clever ideas have been suggested for handling more complicated situations by lossless compression.

The first idea for handling these more complicated patterns is known as *The Same as Earlier Trick*.

The Same As Earlier Trick

Consider you were given the following message to repeat over the telephone:

VJGDNQMYLHKWVJGDNQMYLHADXSGF0VJQGNQMYLHADXSGFVJGNMQMYLHEWADXSGF

Notice that two strings of letters occur several times:

VJGDNQMYLHKWVJGDNQMYLHADXSGF0VJGDNQMYLHADXSGFVJGDNQMYLHEWADXSGF

If you noticed this pattern while transmitting the message, it would save a lot of time to say: *"this part is the same as something I told you earlier."*

To be precise, you would have to point to the start and length of the first occurrence of the string that is being repeated.

The Same As Earlier Trick

VJGDNQMYLHKWVJGDNQMYLHADXSGF0VJGDNQMYLHADXSGFVJGDNQMYLHEWADXSGF

The first 12 characters don't show any repetition so we just have to read them out, *“V, J, G, D, N, Q, M, Y, L, H, K, W”*.

The next 10 characters are the same as earlier ones, so you could say, *“look back 12, copy 10”*.

The next 7 characters are new, so we say *“A, D, X, S, G, F, 0”*.

The next 16 characters are a repeat, so we say *“look back 17, copy 16”*.

Another repeat follows, so we say *“look back 16, copy 10”*.

Two new characters have to be listed, *“E, W”*.

We finish by saying *“look back 18, copy 6”*.

The Same As Earlier Trick

Our original 63 character string was:

VJGDNQMYLHKWVJGDNQMYLHADXSGF0VJGDNQMYLHADXSGFVJGDNQMYLHEWADXSGF

Using the abbreviations **b** for "back" and **c** for "copy", our revised string could be written as:

VJGDNQMYLH-KW-b12c10-ADXSGF-0-b17c16-b16c10-EW-b18c6

Ignoring the dashes, which we inserted for clarity, our message has been shortened from 63 characters to 44, for a compression ratio of 63/44=1.43.

The Same As Earlier Trick

It's easy to come up with a compression scheme if you know beforehand what kind of strings are going to be repeated. For a message in which ”The Supreme Court of the United States” will appear many times, you can write **SCOTUS** with the understanding that your recipient will easily recognize this abbreviation and expand it back to its original form.

But **The Same As Earlier Trick** allows you to compress a message containing **any kind** of repeated string, and does not require that your receiver knows any of the “abbreviations” in advance.

Repeating a short string many times

Here's one extra feature of The Same As Earlier Trick.

Suppose your message was the 16 characters:

FGFGFGFGFGFGFGFG

Then your "compressed" version might be

FG-b2c2-b2c2-b2c2-b2c2-b2c2-b2c2-b2c2

which is longer than the original.

But in fact, you could write just 7 characters (we count the "14" as a single character)

FG-b2c14

achieving a compression ratio of 16/7=2.28. *(Work it out!)*

It's tricky, so let's check it out!

You might not see how the compressed string

FG-b2c14

is able to store the full string:

FGFGFGFGFGFGFGFG

Think about it this way. The compressed string begins by saying:
To uncompress me, start by writing down FG.

Then it says:
fourteen times, copy the letter two positions back to the current position.

Now can you see that this recovers our string?

The Dictionary Trick

The best opportunities for compression come when the message is large, or has repeated patterns that we can recognize.

The Run Length Encoding Trick looked for repetitions that were right next to each other.

The Same As Earlier trick, looked for repetitions that might occur in scattered positions.

In both these examples, the patterns we discovered were somewhat random strings of letters or symbols.

For the Dictionary Trick, the information to be compressed consists of words or chunks of information, and all we are going to do is pay attention to how often and where each word is used. Along the way, we will build a “dictionary” that will help us compress and uncompress the message.

Using a real dictionary

To get an idea of how this might work, let's look at an extreme example, where we agree in advance that we are going to use Webster's Dictionary and replace every word by its numeric order in that dictionary.

The message:

```
"There is a chance to escape by the back door!"
```

might become:

```
259249, 126663, 1, 41107, 262050, 82554, 34352, 258772, 20728, 72347
```

The numeric version of the message can be translated back to words as long as the receiver has a copy of the dictionary. However, it may seem our message has gotten mysterious, but not shorter! (Just look at it!)

Building our own dictionary

The multi-digit numbers seem longer than the words they represent, but in fact the computer can store those numbers much more efficiently than the words.

Also, the numbers in our coded message are very large because we have the choice of any word in the whole dictionary, and English has several hundred thousand words. If we can restrict ourselves to a smaller choice, then the numbers in our coded message will be smaller, and that might help our compression.

One way to get a small dictionary is simply to use the message itself to build the dictionary. This can pay off, especially if the message is short or has a lot of repetition.

To illustrate this idea, let's consider using dictionary compression on a small example, the Gettysburg Address.

The Gettysburg Address - Abraham Lincoln

Four score and seven years ago, our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this. But, in a larger sense, we cannot dedicate - we cannot consecrate - we cannot hallow - this ground. The brave men, living and dead, who struggled here, have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us - that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion - that we here highly resolve that these dead shall not have died in vain - that this nation, under God, shall have a new birth of freedom - and that government of the people, by the people, for the people, shall not perish from the earth.

The Gettysburg Address Dictionary

If we include the title, there are 273 words in the text.

If we include punctuation and blanks, there are about 1,532 characters or bytes in this file, which is how the computer will measure its size.

Let's begin by constructing a "Gettysburg dictionary". We simply list the unique 143 words, in alphabetic order:

1	a
2	above
3	abraham
4	add
5	address
...	
139	who
140	will
141	work
142	world
143	years

Encode the message using the dictionary

To compress the message, we replace each word by its dictionary index:

```
the gettysburg address abraham lincoln four score and seven years ag
122 58         5       3       75      53   111   10  113   143   7
```

and now the encoded message even **looks** shorter than the original.

In fact, because each index value is less than 256, each number you see can be stored by the computer as efficiently as a single letter.

This means that the encoded version of the Gettysburg Address will shrink from 1,532 characters (=bytes = numbers 0:255) down to 273 bytes, for a compression ratio of 1532/273=5.6, a very significant savings.

Decode the message using the dictionary

To uncompress the message, the receiver replaces each dictionary index by the corresponding word:

```
122 58         5       3       75      53   111   10  113   143   7
the gettysburg address abraham lincoln four score and seven years ag
```

Now we've left out one important detail...to uncompress the file, the receiver needs not just the compressed message but also **the dictionary that was created**.

In fact, for large messages, or large sets of messages that use the same small “dictionary” of words, the dictionary turns out to be a bargain.

If you think about it, this is similar to the Shipper's Code we discussed earlier, in which a message is compressed to a single number, which is a word in the Shipper's Code dictionary, and then decompressed by someone who also has a copy of that dictionary. The difference in our example is that now we can build a new dictionary to suit our needs.

Socrative Quiz PartVI_Quiz1

IUZGAZ34E

1. If the compression ratio is 1, then the compression program didn't compress the file at all. (True, False)
2. Run Length Encoding (RLE) can compress a message if it contains repeated sequences of letters (True, False)
3. Because a black and white image is a picture of a real physical object, it cannot be compressed (True, False)
4. The Same As Earlier Trick is better than Run Length Encoding because it can notice repeated patterns that are not next to each other. (True, False)
5. Run Length Encoding is a lossy compression scheme; if the compressed file is uncompressed, some data will have been lost. (True, False)

How the computer stores data

In order to understand the next approach to compression, it is necessary to be a little more realistic about how information is stored in the computer.

When the user enters letters like $\mathbf{a}$, $\mathbf{b}$ or $\mathbf{c}$, the computer doesn't actually store these letters. The computer can only store numbers, and so it uses a table that can "translate" letters and other symbols to a numeric code, or convert the numeric code back to a letter.

For example, the letter $\mathbf{a}$ might be represented by the numeric code $\mathbf{27}$, $\mathbf{b}$ by $\mathbf{28}$, $\mathbf{c}$ by $\mathbf{29}$ and so on.

When the user enters a character such as $\mathbf{a}$, we say it is encoded into the numeric symbol $\mathbf{27}$, and when the computer sends the numeric symbol $\mathbf{27}$ back to the user, it is decoded as $\mathbf{a}$.

$$a \rightarrow \text{(encode)} \rightarrow 27 \rightarrow \text{(decode)} \rightarrow a$$

A table is used to translate between humans and computers

This means that a string which the user enters, such as **cab**, will be stored in the computer as a string of numbers, namely **29,27,28**.

Computers do not use the decimal system, but instead a base two system called *binary arithmetic*.

In the following example, we'll use base 10 for simplicity, but the same thing will happen for the computer's binary system.

Now we look more closely at how a computer handles characters. The following example table *(not the actual table used in computers!)* contains numeric codes for 100 symbols. These codes run from 0 to 99, that is, they are each a one- or two-digit decimal number.

The list includes the alphabetic characters in lower and uppercase, punctuation marks, other symbols, and some accented letters that occur in foreign languages.

space	0	T	20	n	40	(	60	á	80
A	01	U	21	o	41	)	61	à	81
B	02	V	22	p	42	*	62	é	82
C	03	W	23	q	43	+	63	è	83
D	04	X	24	r	44	,	64	í	84
E	05	Y	25	s	45	-	65	ì	85
F	06	Z	26	t	46	.	66	ó	86
G	07	a	27	u	47	/	67	ò	87
H	08	b	28	v	48	:	68	ú	88
I	09	c	29	w	49	;	69	ù	89
J	10	d	30	x	50	<	70	Á	90
K	11	e	31	y	51	=	71	À	91
L	12	f	32	z	52	>	72	É	92
M	13	g	33	!	53	?	73	È	93
N	14	h	34	”	54	{	74	Í	94
O	15	i	35	#	55	|	75	Ì	95
P	16	j	36	$	56	}	76	Ó	96
Q	17	k	37	%	57	_	77	Ò	97
R	18	l	38	&	58	Ø	78	Ú	98
S	19	m	39	’	59	ø	79	Ù	99

How human text is stored as computer data

Let’s look at how an example sentence is encoded by the table.

The 23 character English sentence **“Meet your fiancé there.”** would become the following list of numeric codes:

M	e	e	t		y	o	u	r		f	i	a	n	c	e’		t	h	e	r	e	.
13	31	31	46	00	51	41	47	44	00	32	35	27	40	29	82	00	46	34	31	44	31	66

Since we’re concerned about compression, it’s important to realize that, when measuring the length of the list of numeric codes, we should not include any spaces or commas. As far as the computer is concerned, the message has become the following string of 46 characters:

```
1331314600514147440032352740298200463431443166
```

Since each numeric code is exactly two digits, we can easily break this string into its 23 separate codes if we need to.

How computer data is translated back into human text

All the numeric codes used two digits, so $\mathbf{A}$ is coded as $\mathbf{01}$, not $\mathbf{1}$.

Because of this, when we see a coded string like

```
1331314600514147440032352740298200463431443166
```

we only break it up into digit pairs to translate it back to text:

```
13 31 31 46 00 51 ...
```

and we would not break it up into, say:

```
1 33 13 14 60 0 5 1...
```

The translation between characters and numeric codes must always have a single interpretation, even when the numeric codes are written packed together, with no spaces or commas between them.

This rule will become an important issue shortly!

The Shorter Symbol Trick

Now we are ready to discuss the ideas behind the next compression procedure, known as **The Shorter Symbol Trick**. Actually, this trick is based on something we do all the time in everyday communication. The idea is that if you use a phrase often, it's worthwhile to come up with a shorter version of it.

Everyone knows that **USA** is short for *The United States of America*; we save a lot of time or typing by using the 3 letter abbreviation for this 24 letter phrase.

The sky is blue in color is another 24 letter phrase, but no one has bothered to come up with an abbreviation for it.

What is the difference? One phrase is rarely used; the other occurs often, so inventing and using an abbreviation for it is worth the effort.

Prioritize the common symbols

Let's see if we can apply the Shorter Symbol Trick to compress the message we considered earlier, *"Meet your fiancé there."* We know we should focus on the most commonly occurring items.

Originally, we used two digits to represent every number, so even 0 through 9 were written as $\mathbf{00}$ through $\mathbf{09}$. But what if we had the option of including some 1-digit codes, namely But we could certainly save space by writing them as single digits $\mathbf{0}$ through $\mathbf{9}$. Now we have 10 of our symbols that only take one digit to write rather than two.

What's the best way to take advantage of our shorter symbols?

Prioritize the common symbols

e and t are the most common in English, but in the original table we used two digits for each of them.

Now that we have 10 1-digit codes, how about rearranging our table so short codes go with common letters? Suppose $\mathbf{e}$ is coded as $\mathbf{8}$ and $\mathbf{t}$ as $\mathbf{9}$.

Now we cut down the encoded message from 46 to 40 decimal digits (compression ratio = 46/40=1.15):

M	e	e	t		y	o	u	r		f	i	a	n	c	e		t	h	e	r	e	.
13	8	8	9	00	51	41	47	44	00	32	35	27	40	29	82	00	9	34	8	44	8	66

and we assume that in the computer the message is stored as one long string:

1388900514147440032352740298200934844866

How does the receiver decode the message?

The person sending the message is happy, because now the encoded message has been compressed...but we have created a serious problem for the person receiving the message, who has to do the decoding.

Since we switched to using both 1-digit and 2-digit numeric codes, it's no longer clear how to chop up the message into the individual codes. Our message is:

`138890051414744003235274029820093484 4866`

Let's concentrate on the first five digits (13889). These could be broken up into the codes **13 8 8 9** or **13 88 9** or **13 8 89** which would decode to **Meet** or **Mút** or **Meù**.

There is no way to tell which of these three messages is the intended one.

Our scheme has shortened the message, but now we can't decode it!

A modified table fixes the problem

We can save the Shorter Symbol Trick if we make some symbols longer.

One way to solve our problem is to put a $\mathbf{7}$ in front of every one of the ambiguous 2-digit codes.

This will allow us to have short (1 digit) symbols for $\mathbf{e}$ and $\mathbf{t}$, medium (2 digit) symbols for other alphabetic characters, and long (3 digit) symbols for rarely used characters.

Our new coding table includes all the changes:

space	00	T	20	n	40	(	60	á	780
A	01	U	21	o	41	)	61	à	781
B	02	V	22	p	42	*	62	é	782
C	03	W	23	q	43	+	63	è	783
D	04	X	24	r	44	,	64	í	784
E	05	Y	25	s	45	-	65	ì	785
F	06	Z	26	t	9	.	66	ó	786
G	07	a	27	u	47	/	67	ò	787
H	08	b	28	v	48	:	68	ú	788
I	09	c	29	w	49	;	69	ù	789
J	10	d	30	x	50	<	770	Á	790
K	11	e	8	y	51	=	771	À	791
L	12	f	32	z	52	>	772	É	792
M	13	g	33	!	53	?	773	È	793
N	14	h	34	”	54	{	774	Í	794
O	15	i	35	#	55	|	775	Ì	795
P	16	j	36	$	56	}	776	Ó	796
Q	17	k	37	%	57	_	777	Ò	797
R	18	l	38	&	58	Ø	778	Ú	798
S	19	m	39	’	59	ø	779	Ù	799

The Shorter Symbol Trick

Using the new table, our message becomes 41 characters long:

13889005141474400323527402978200934844866

but now there is only one way to decode the message:

```
13 8 8 9 00 51 41 47 44 00 32 35 27 40 29 782 00 9 34 8 44 8 66
 M e e t     y  o  u  r     f  i  a  n  c  e’    t  h e  r e  .
```

The original message used 46 characters and now we’re down to 41 (compression ratio=46/41=1.21). In practice, the compression ratios are much better.

The text book, for instance, requires about half a million characters of storage. However, using just the two compression tricks we have described, the compressed version is about 160,000 characters, for a compression factor greater than 3.

ZIP files use the Shorter Symbol Trick

One example of how the tricks we have discussed are used in real life is when a lossless ZIP file is created:

1. the original file is transformed using the Same As Earlier trick;
2. the transformed file is analyzed to see which symbols occur most frequently;
3. using the Shorter Symbol Trick, a coding table is created in which frequent symbols get shorter codes;
4. the transformed file is encoded with the new coding table.

To expand the ZIP file, these steps are undone in the reverse order using the UNZIP program.

Compress a dictionary

The file **dictionary.txt** is 3.15 million characters in size:

```
ls dictionary.txt
-rwxrwxrwx 1 jburkardt staff 3151520 May 29 2013 dictionary.txt
```

```
zip dictionary.zip dictionary.txt
ls dictionary.zip
-rw-r--r-- 1 jburkardt staff 896590 Apr 26 10:44 dictionary.zip
```

ZIP compresses the file to less than 1 million characters. Our compression ratio is $3141520/896590 = 3.51$

```
unzip dictionary.zip
ls dictionary.txt
-rwxrwxrwx 1 jburkardt staff 3151520 May 29 2013 dictionary.txt
```

UNZIP recovers the exact original text, another lossless compression.

Socrative Quiz PartVI_Practice_Quiz2

IUZGAZ34E

1. Every letter, symbol, or digit entered into a computer is actually stored as some kind of binary number (True, False)
2. What kind of compression scheme replaces the string AAABCCCC by “3A, 1B, 4C”? (Run Length Encoding, ZIP, Shorter Symbol Trick, Telegraph Code)
3. The Shorter Symbol Trick works by: (using a list of abbreviations for common words, using short symbols for frequently used words, replacing every letter by a number)
4. To make sure that a string compressed by the Shorter Symbol Trick could be uncompressed again, we had to: (make some symbols for rarely used words longer, add new symbols, put a special code between every pair of symbols)
5. An example of the Shorter Symbol Trick in real life would be replacing frequent occurrences of “President of the United States” by **POTUS** (True, False).

So...what about lossy compression?

We have been considering lossless compression, that is, methods for squeezing a message into a smaller version, which can later be expanded to recover an exact copy of the original information.

Another option is available, called lossy compression. The name is chosen to indicate that, when using such a method to compress a file, when the compression is reversed, *some of the original information has been permanently lost.*

Lossy compression is almost <u>never</u> used on text files. If my decompressed dictionary was missing information, I'd be very upset!

Here's a simple example of lossy compression, where it turns out we can actually be pretty sure of what the original was.

Sort of an example of lossy compression

Howard Beale's monolog in "Network", compressed by removing $\mathbf{a}$, $\mathbf{e}$, $\mathbf{i}$, $\mathbf{o}$, $\mathbf{u}$:

```
dnt hv t tll y thngs r bd.
vrybdy knws thngs r bd
ts dprssn.
vrybdys t f wrk r scrd f lsng thr jb.
Th dllr bys nckls wrth; bnks r gng bst;
shpkprs kp gn ndr the cntr;
pnks r rnnng wld n th strt,
nd thrs nbdy nywhr wh sms t knw wht t d,
nd thrs n nd t t.
```

You can probably work out most of the original text, but that's only because you know English, and you can get a sense of what is being said!

Sort of an example of lossy compression

Howard Beale's monolog in "Network":

I don't have to tell you things are bad.
Everybody knows things are bad.
It's a depression.
Everybody's out of work or scared of losing their job.
The dollar buys a nickel's worth; banks are going bust;
shopkeepers keep a gun under the counter;
punks are running wild in the street,
and there's nobody anywhere who seems to know what to do,
and there's no end to it.

Lossy compression works great for many kinds of data

Although text is not appropriate for lossy compression, it turns out that many other kinds of data files are ideal candidates for lossy compression.

The reason is that a typical computer data file may contain a mixture of useful and useless information. For example, a simple way to record sound involves taking 44,100 samples per second.

A typical CD with 640 Megabytes capacity can hold:

- 1 hour of uncompressed music, or
- 2 hours of losslessly compressed music,or
- 7 hours of music compressed using the lossy MP3 compression.

An MP3 recording is decompressed as the CD is played. The resulting music will not be identical to the original recording, but is usually too close to notice any difference.

How does MP3 compress a sound file?

The techniques used by MP3 to compress audio recordings are based in part on recognizing the limitations of the human ear:

- you can't hear very low or high frequency tones;
- you can't hear very soft tones;
- you can't hear soft tones just after a loud one;
- if low and high frequency tones are played with the same loudness, you hear the low frequency (drumbeat over flute);
- if music is to be played in a noisy environment, only loud or low frequency information will be heard at all.

Using ideas like this, a compression program can remove as much as 80 or 85% of the original information, producing a stripped-down recording that will still sound very close to the original.

The Leave-It-Out Trick

MP3 compression uses the idea that some of the information in an audio signal is not very important. If the listener will not be able to detect whether this data is there or not, then we can delete it, and decrease the size of our data.

This idea can be called the Leave-It-Out-Trick. Of course, if we actually delete part of our data as part of our compression process, we won't be able to recover it when the data is decompressed. So this technique is only used in lossy compression methods.

Lossy compression relies, in part, on the idea that electronic recording devices pick up and store lots of information that we humans don't actually need, somewhat like color TV for dogs.

Lossy compression works for visual data too

Photos, drawings, and movies can also be compressed, and again, compression is aided by understanding limitations of the human eye, including:

- just as the human ear can't hear high frequency tones, the human eye is not good at detecting high frequency variations in an image, for example, a pattern of alternating black and white lines, if thin enough, will simply register as solid gray.
- the eye essentially averages the information in a small neighborhood around any spot in a picture. Thus, we can blur the image somewhat, without noticeable effect.
- although a computer can display millions of different colors, the human eye is not very sensitive to small color differences;

The JPEG format is a common lossy method for compressing pictures.

How is lossy compression of images possible?

While programs like JPEG use very sophisticated techniques to compress image files, we can easily demonstrate some of the simplest ones. For example, we have **The Leave It Out Trick**.

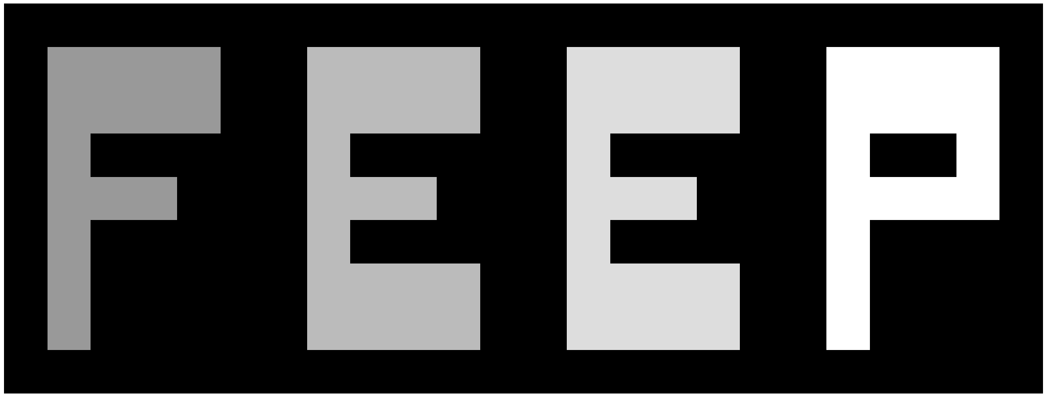
We will look at this method in terms of a black and white image. Note that most such images are actually black and white *and hundreds of shades of gray*, and so are sometimes called grayscale images.

An uncompressed image is stored as a rectangular table, like a piece of graph paper. Each entry in the table, or box in the graph paper, is called a pixel (an abbreviation of *picture element*), and contains a number representing the shade at that point in the picture.

FEEP, a simple image

Here is a simple uncompressed grayscale image file, (called **FEEP**), using gray shades between 0 (black) and 15 (white):

```
0 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 0  0  0  0  0  0  0
0 9  9  9  9  0  0 11 11 11 11  0  0 13 13 13 13 0  0 15 15 15 15  0
0 9  9  9  9  0  0 11 11 11 11  0  0 13 13 13 13 0  0 15 15 15 15  0
0 9  0  0  0  0  0 11  0  0  0  0  0 13  0  0  0 0  0 15  0  0 15  0
0 9  9  9  0  0  0 11 11 11  0  0  0 13 13 13  0 0  0 15 15 15 15  0
0 9  0  0  0  0  0 11  0  0  0  0  0 13  0  0  0 0  0 15  0  0  0  0
0 9  0  0  0  0  0 11 11 11 11  0  0 13 13 13 13 0  0 15  0  0  0  0
0 9  0  0  0  0  0 11 11 11 11  0  0 13 13 13 13 0  0 15  0  0  0  0
0 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 0  0  0  0  0  0  0
```



Compressing FEEP

The FEEP file used a table of 9 rows by 24 columns, containing grayscale values between 0 and 15.

When we display the FEEP picture, we see the regions of darkness and light corresponding to the numeric values.

However, the image does not seem to be made of squares of solid colors.

This is because, in order to be viewed, we must decode the image file. Most programs that decode or display an image file will automatically try to smooth out the rectangular boundaries between various pixels of different colors, to make the picture look better.

Thus, it is actually a little difficult to force our blocky picture to show up as a bunch of blocks!

Cameras create huge images

A more typical grayscale graphics file might use 256 shades of gray, going from 0 for full black to 255 for full white.

Computer generated pictures often have height to width ratios of 4 to 3, (portrait) or 3 to 4 (landscape)

A modern camera or cellphone may store images as a pixel table of 4,000 rows by 3,000 columns, for a total of 12,000,000 pixels.

Many television screens have an array of 1920 pixels wide and 1080 pixels high, or 2,073,000 total pixels.

Many images on computers and the Internet are somewhat smaller, with a typical size being 480 x 360 $=$ 165,600 pixels,

The total number of pixels is known as the resolution of the picture.

We can discard some data from images

Now 12 million gray dots is a lot of information, and your eye might be perfectly satisfied with a much simplified version.

In fact, if I want to send a picture from my phone to someone else, the phone suggests sending a medium or low resolution version, because the picture will still be good enough to view, and I can reduce my data transmission charge by sending a smaller image.

How is a lower resolution version created?

Suppose that we started with a 460 by 360 pixel image file, and simply removed every other row and column. Our example would drop from 165,600 pixels to 230 * 180 = 41,400 pixels, that is, it would be reduced to $1/4$ the number of pixels.

This compression method is easy...and definitely lossy.

But will it result in a usable picture?

A simple example of image compression

Let's experiment by starting with a standard size image, and cutting out the even rows and even columns.

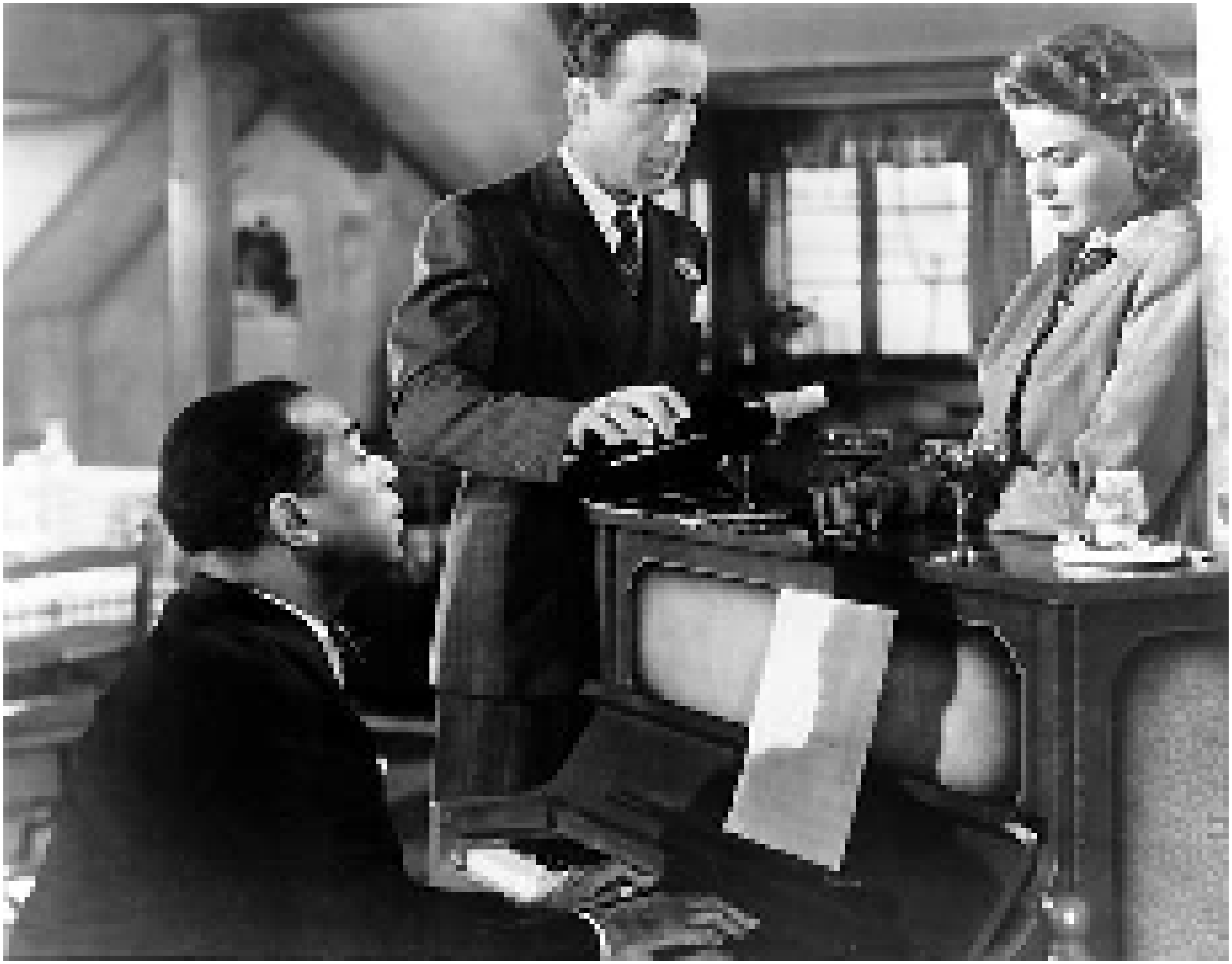
This creates a file that is 1/4 the size of the original, by permanently losing 3/4 of the information.

Our only hope is that, because the missing information is right next to information we are keeping, the eye will not notice the small disruptions.

And if we are satisfied with the compressed image, we can try a second compression....

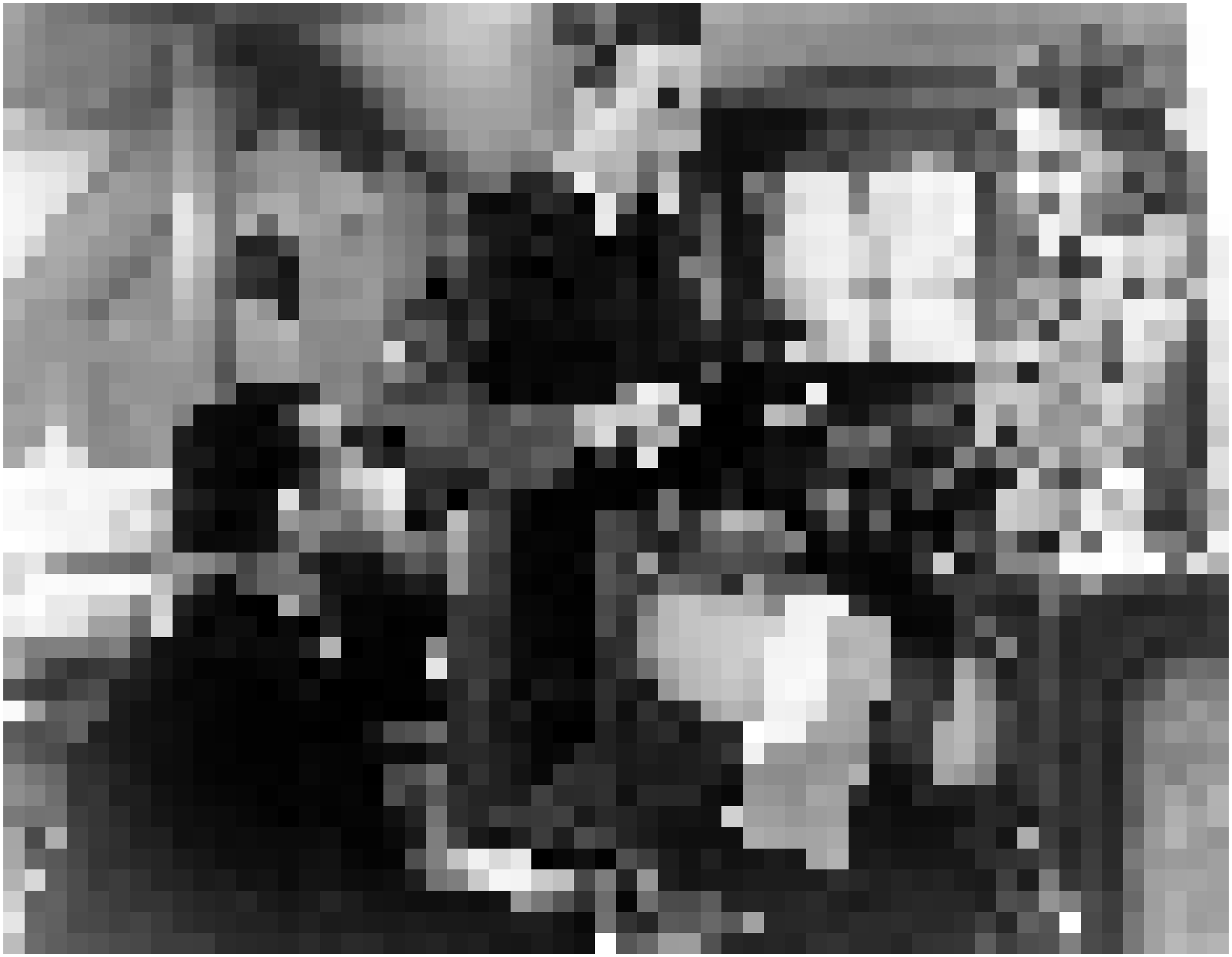
460 * 360 = 165,600 pixels

230 * 180 = 41,400 pixels

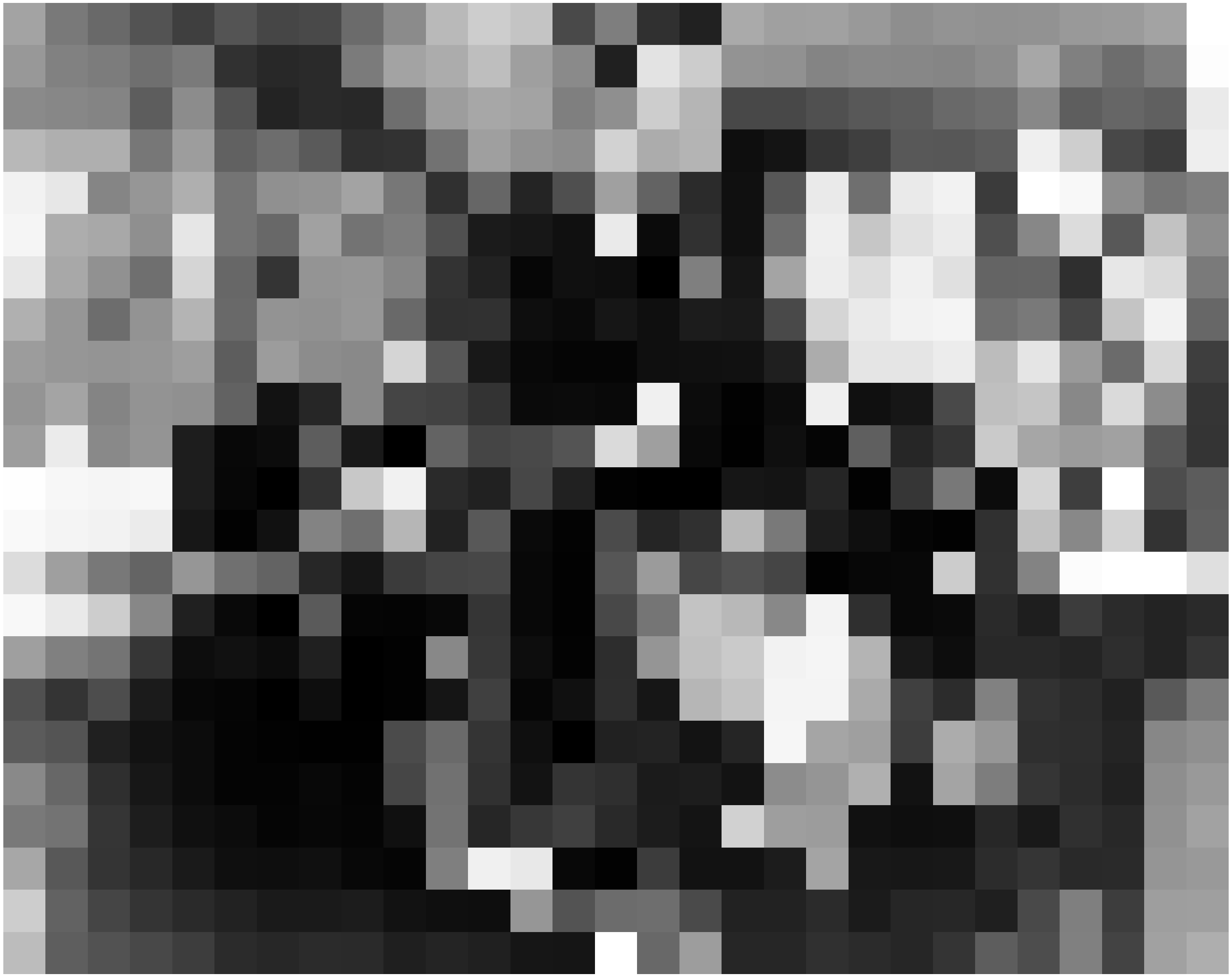


115 * 90 = 10,350 pixels

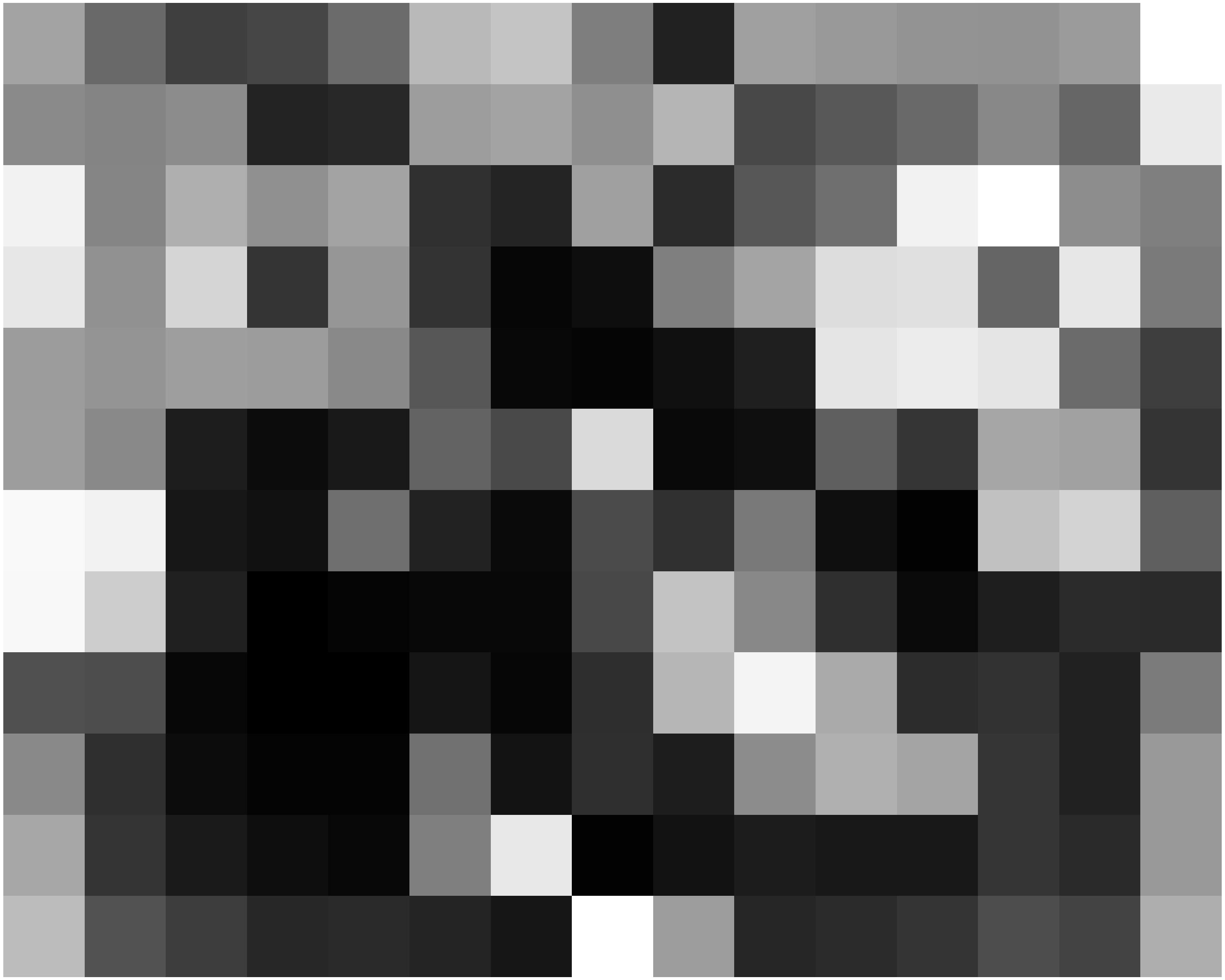
58 * 45 = 2,610 pixels



29 * 23 = 667 pixels



15 * 12 = 180 pixels



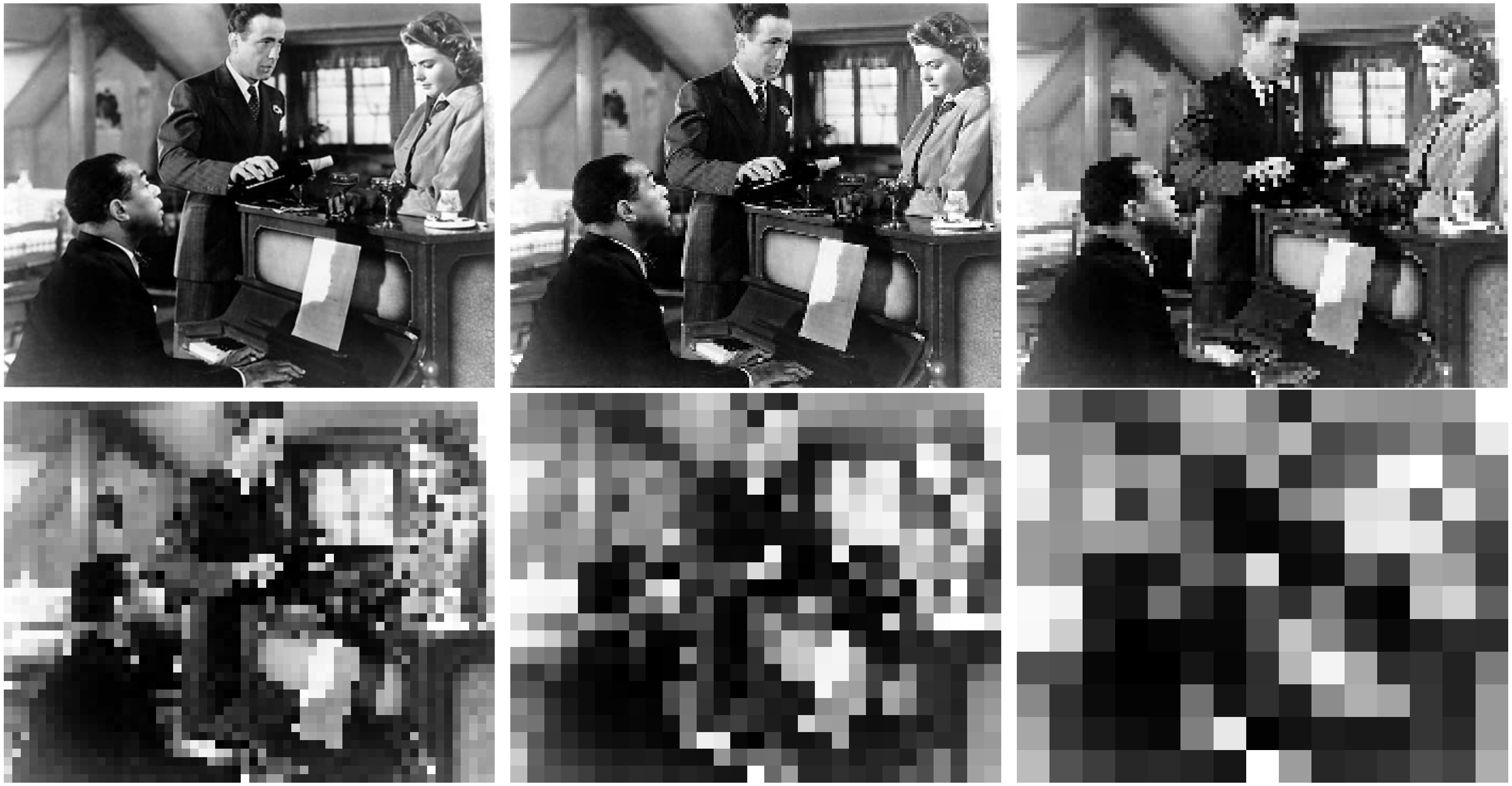
We can throw away some (not all!) of the data

If we look at all six versions of our image together, even the second compressed version may not look too bad...because we've reduced its size to show it together with the others.

But if we go back to the larger version, our eye may be able to spot some unnatural looking jagged boundaries that correspond to the rectangular pixels that make up the computer image.

But even after one or two more compressions, we can still see the main items in the picture. Even with only 667 pixels, we may not be able to tell what is going on, but we can guess that there are 3 or 4 people in the picture. Of course, with 180 pixels, there is just not enough information for our eyes to make any sense of things.

We can throw away some (not all!) of the data



JPEG is a lossy compression format for images

We have applied the Leave It Out Trick in a very simple manner, by dropping rows and columns of the image file without worrying about the information they contain.

Because images are used so frequently, and the raw images created by cameras and medicals devices are so large, it has become important to develop much better lossy compression techniques for squeezing out as much information as possible, while retaining the information necessary for the eye to “read” the picture correctly.

The JPEG format is a widely used image compression technique that uses the Leave It Out Trick in a more intelligent and careful way.

JPEG stores the full image efficiently

JPEG files are more complicated than the simple image file format we discussed earlier; it is enough to know that the file size is measured in characters. If we convert our 480x360 pixel image to the standard JPEG format, it is stored using 60,231 characters, rather than 165,600 pixels.

Although 60,231 is smaller than 165,600, we have not actually asked JPEG to compress the image information at all.

The number of characters is smaller, only because JPEG knows how to store several pixels into a single character.

JPEG compresses by looking for patterns

To compress the image, JPEG divides up the pixels into small squares of 8x8 pixels called blocks, and looks for patterns.

If all 64 pixels in a block are the same color, than JPEG can simply "leave out" 63 numbers, remembering that the whole block is one shade.

If the 64 pixels in a block are close to some color, then the computer can consider averaging the colors and using that average for the entire block.

If the colors in the block change smoothly from the left to the right, then the computer can store the left and righthand colors, and remember that the pixels in between get in between shades.

There are other patterns of variation that are simpler to describe than simply recording the individual shades of all 64 pixels.

We can control the amount of JPEG compression

We may allow JPEG to replace the exact data by a pattern even if the pattern is only an approximate match.

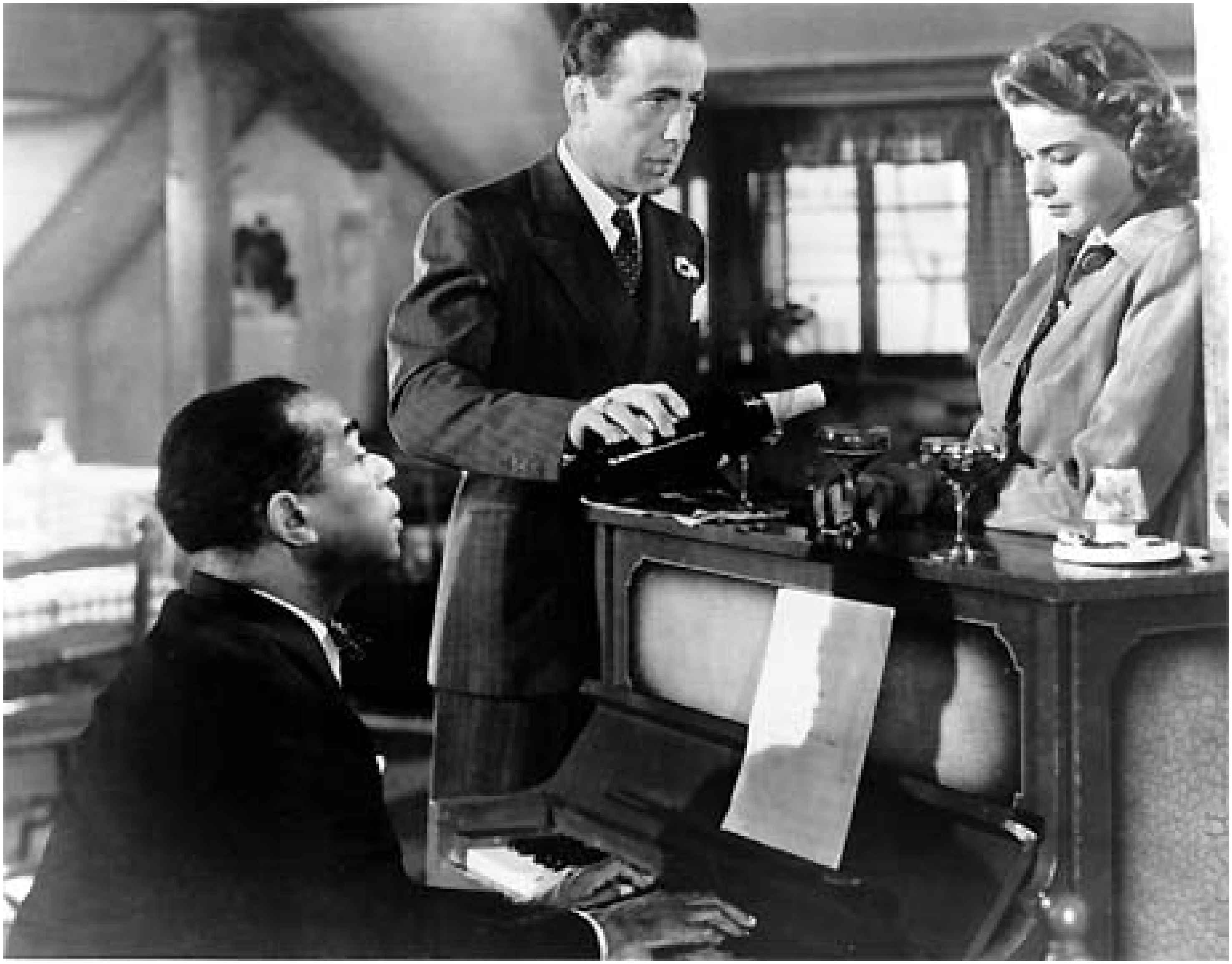
The quality of a JPEG image is 100% if we never allow any such approximation; in this case, JPEG compression is lossless.

As we reduce the quality, we are allowing JPEG more leeway in replacing exact picture data with approximate patterns. As we reduce the quality, we improve the chances for compression.

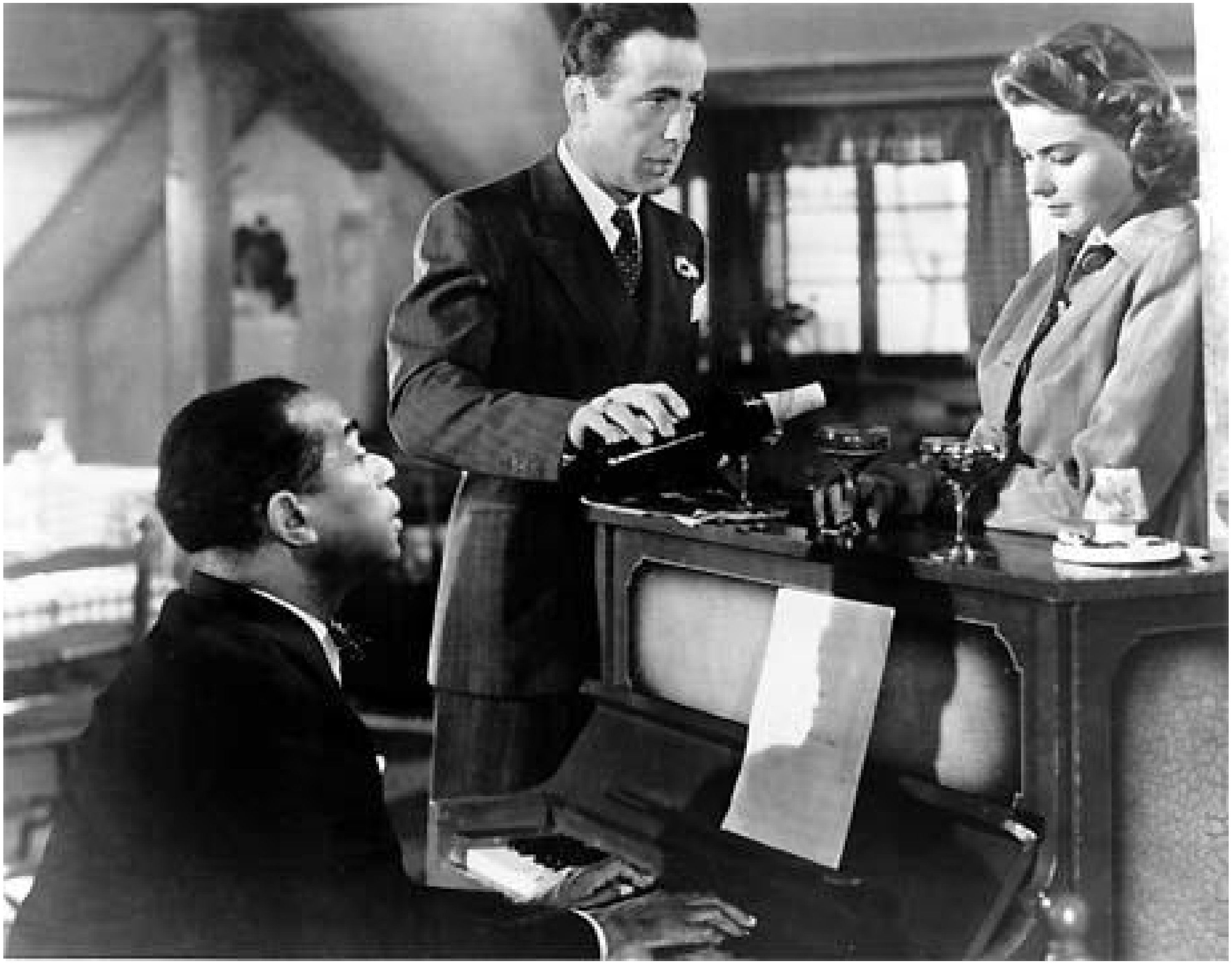
We will see that JPEG can sometimes reduce the size of an image file by a compression ratio of 20 without the eye noticing much difference.

Here, we will repeat our exercise with the black and white image, decreasing the requested JPEG quality from 100% down to 1%.

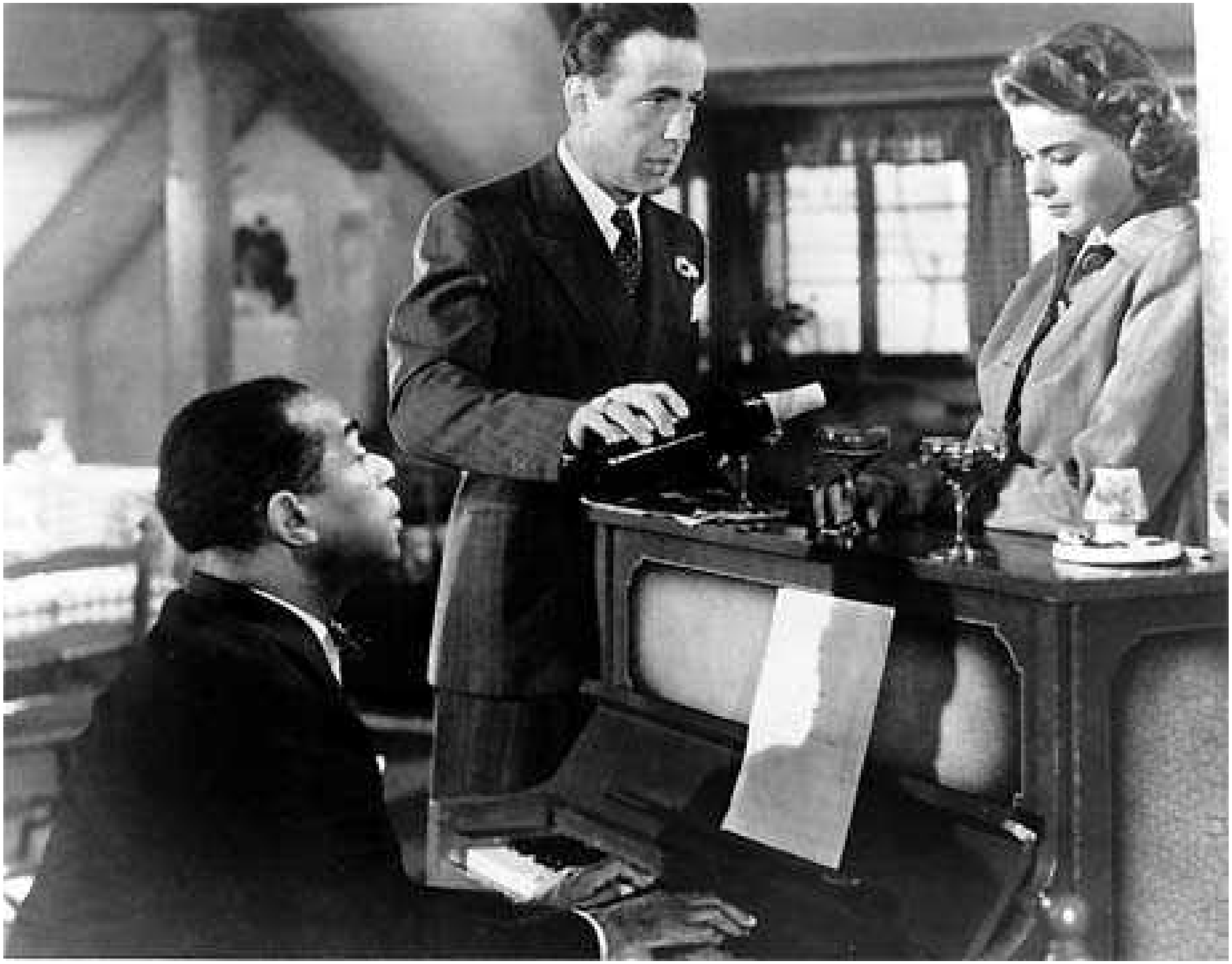
JPEG 100% Quality = 60,231 characters



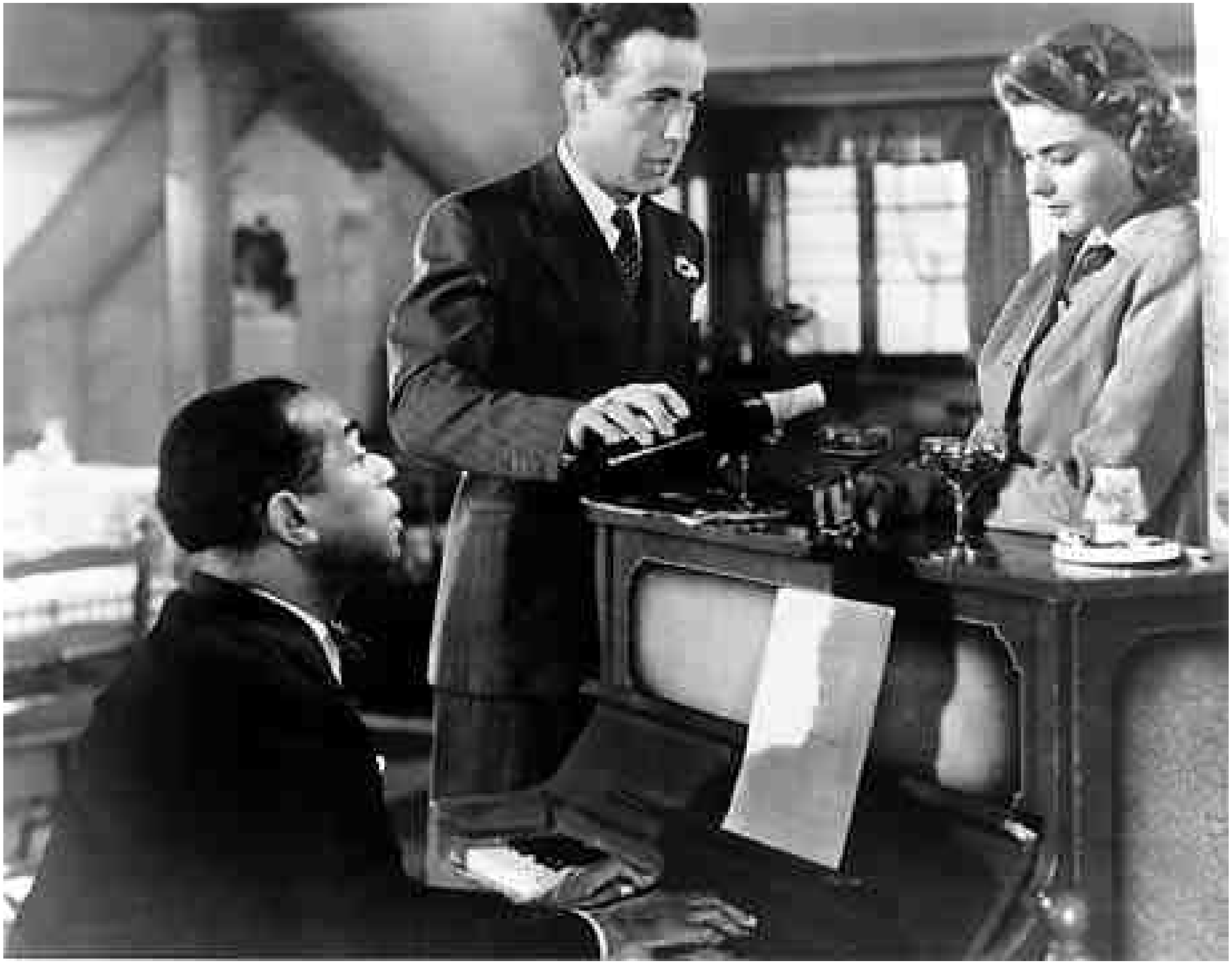
JPEG 75% Quality = 26,878 characters



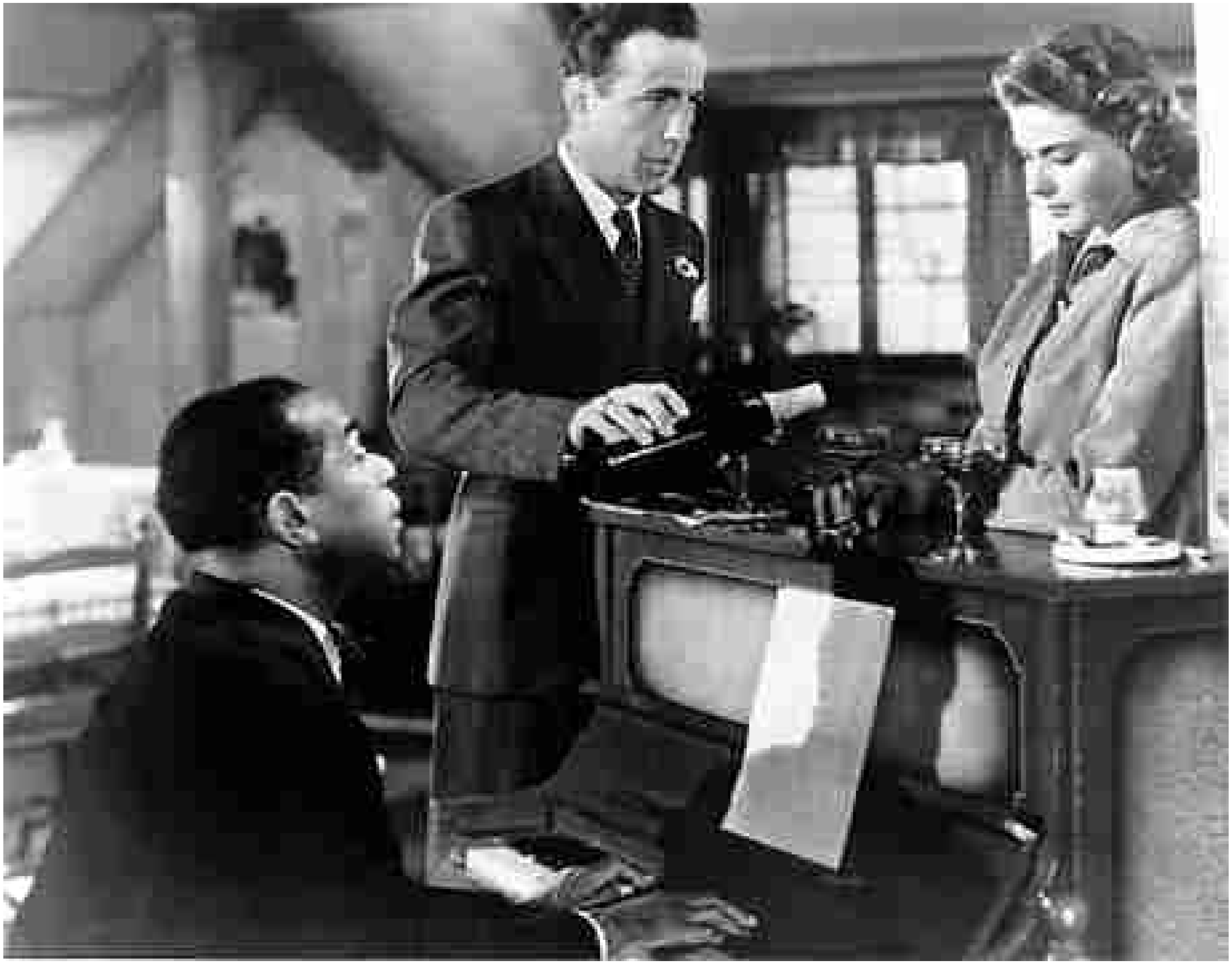
JPEG 50% Quality = 21,045 characters



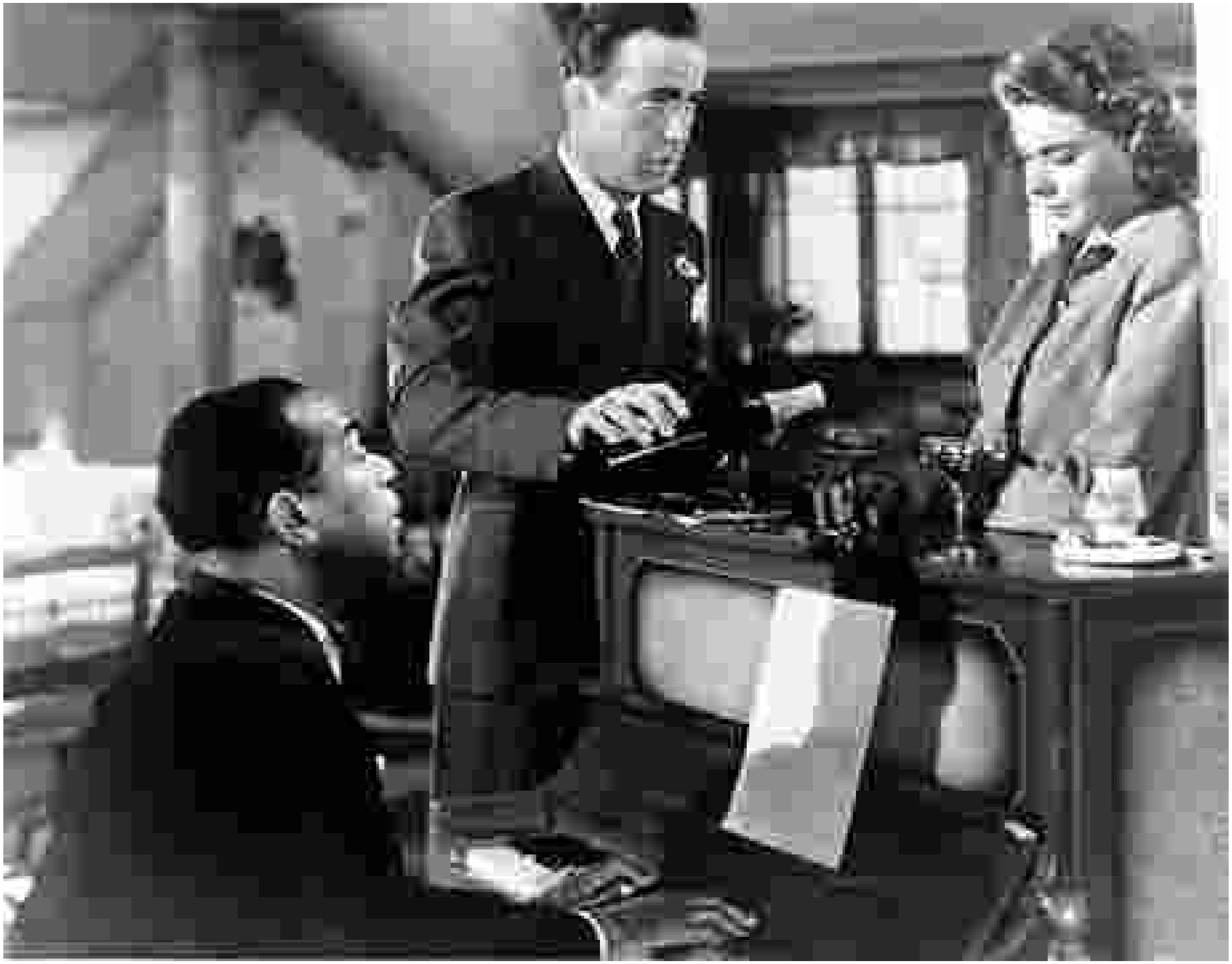
JPEG 25% Quality = 12,520 characters



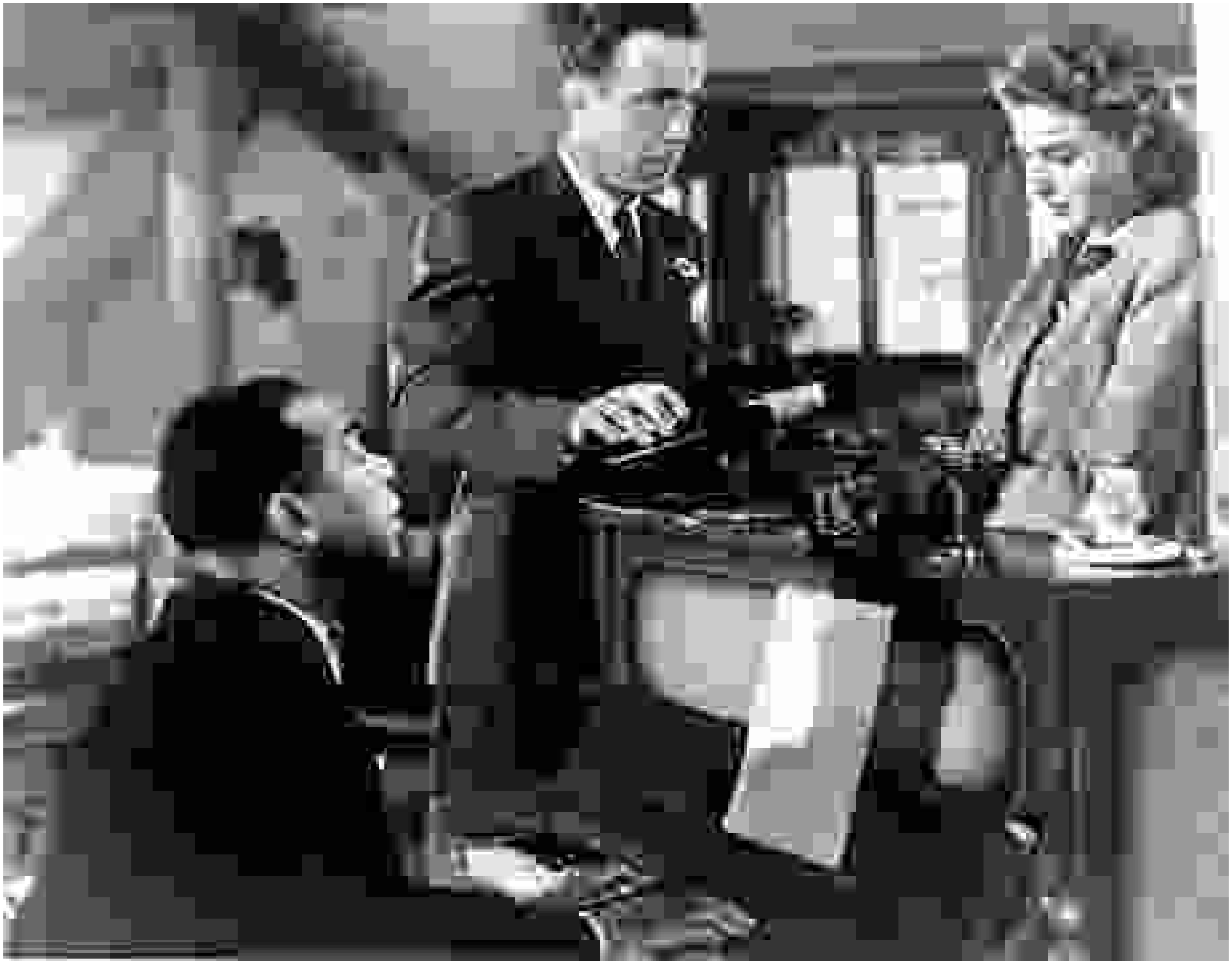
JPEG 16% Quality = 9,482 characters



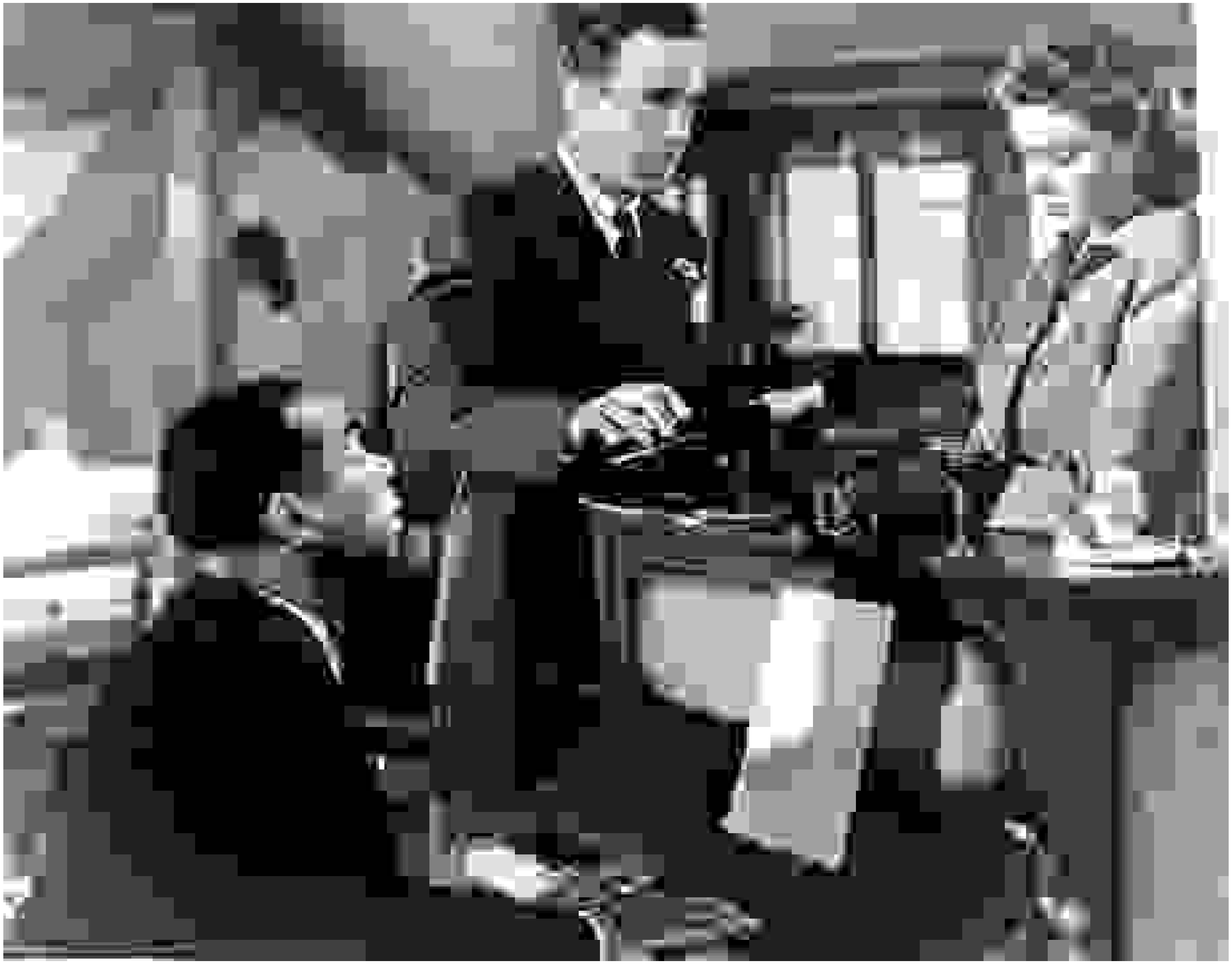
JPEG 8% Quality = 5,711 characters



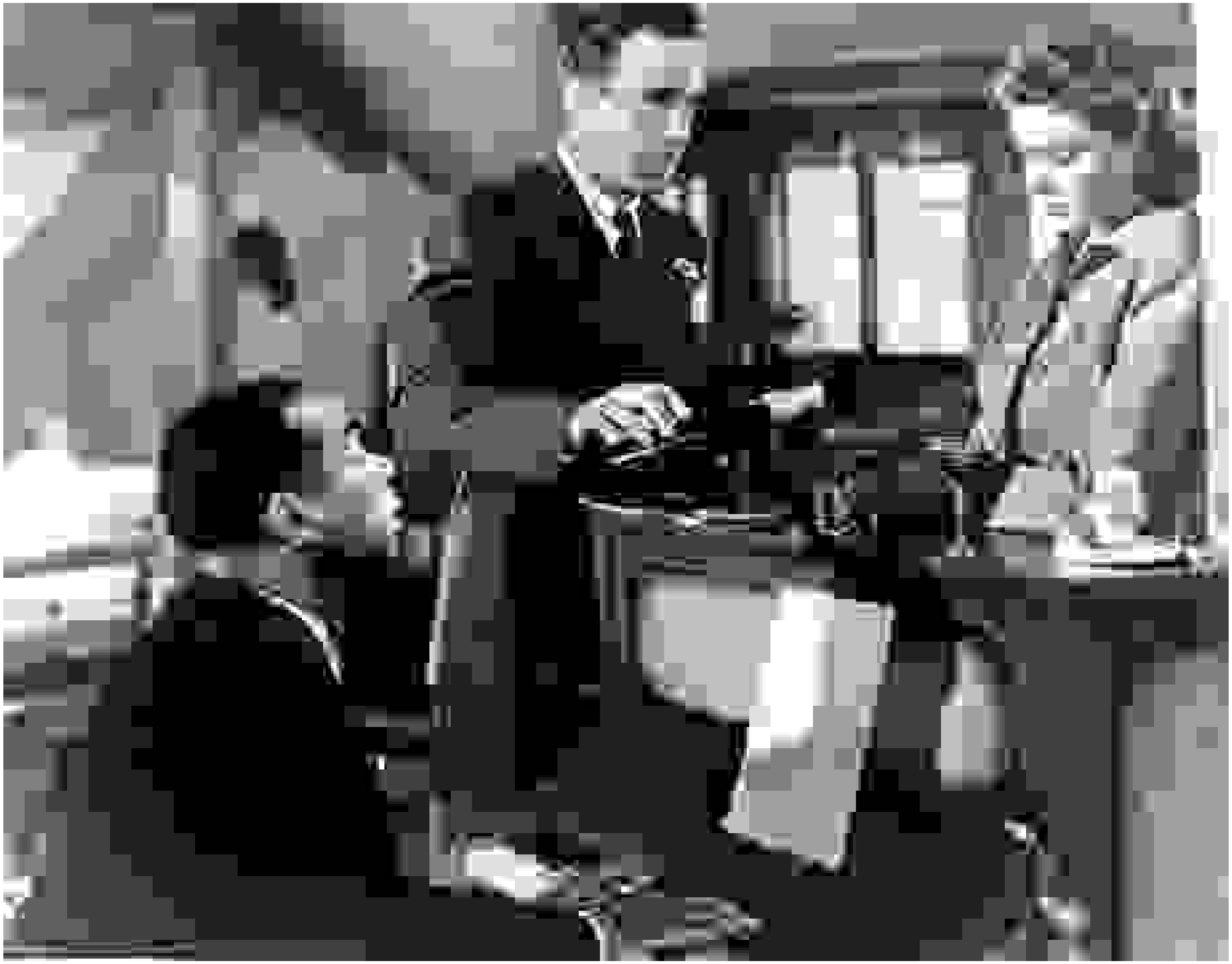
JPEG 4% Quality = 3,229 characters



JPEG 2% Quality = 2,265 characters



JPEG 1% Quality = 1,261 characters



Compare the original to the compressed versions

Using JPEG, we started with our file at 100% quality and 60,231 characters and reduced it to 8% quality and 5,711 characters, for a compression factor of 12, without much noticeable loss of information.

Even going to 1% quality, 1,261 characters and a compression factor of 50, the picture is recognizable.

What is perhaps more surprising is what happens when we try to show all 9 versions of the image together. Because we reduce the size of each image, the 1% image seems about as good as the 100% image. This is because, as we said before, the eye is not good at seeing small details. When you reduce an image, only the large details remain, and we can not longer notice that the small details in the 100% picture are sharp, while the small details in the 1% picture are very blocky.

Audio compression

Stereo sound is stored by recording the amplitude at the left and right microphones 44,100 times each second. Each amplitude is a 16 bit or 2 byte piece of data, so a minute of stereo sound creates

$$60 * 44100 * 2 * 2 \approx \text{ 21 MegaBytes of raw data.}$$

and so an hour of raw music data would be about 1.2 GigaBytes.

Audio compression formats like MP3 and AAC look for patterns in the data that can be replaced by "abbreviations", and for discardable information, such as sounds that are too soft, too high, or too rapid, for the human ear to detect.

By processing the sound data in this way, a compression factor of as much as 20 can be achieved with little noticeable loss at playback time, so an hour of music might be compressed to 60 MegaBytes.

WAV, a common audio format on Windows machines, does not use compression.

Video compression

Videos, that is, movies, animations, YouTube clips and television broadcasts, contain both sound and visual information.

The sound information can be compressed as in a regular music file.

However, the video information must be compressed more severely than in a typical image. This is because there is so much of it. A typical movie or animation must display in each second of play a sequence of about 30 fresh images.

Let's assume that a single uncompressed color image is about a million pixels, which works out to about 3 million bytes, or 3 MegaBytes of computer memory. An uncompressed movie lasting two hours would require about

$$2 * 60 * 60 * 30 * 3 \approx 650 \text{ GigaBytes of video storage}$$

which would take up all or most of your computer hard drive. Instead, we expect a typical movie to be closer to 5 or 6 GigaBytes in size, so the compression factor may be as high as 100.

Video compression

Compression of a movie starts by compressing each individual frame of the movie, just as we saw for a still image or photograph.

But in animations we have another way to achieve a huge amount of compression because usually, from frame to frame, most of the image stays the same. In a movie where a character is standing in front of a house talking, the location of the house, the lawn, the trees and the sky don't change much or at all from frame to frame. Video compression methods carefully compare one frame to the next. Wherever possible, they describe the next frame as "the same as this frame...except" and then list the parts that change.

A well known video compression format is MPEG4, which is used by DivX, Xvid and other such video display programs.

Note that WAV, a common audio format on Windows machines, does not use compression.

Conclusion

Compression is a technique that allows us to use to maximum benefit our limited resources (computer memory, network bandwidth, disk space, CD or DVD storage capacity).

Some compression methods are lossless; whenever we reverse the compression process, we get back exactly the original data. Lossless compression relies on techniques such as The-Same-As-Earlier Trick and the Shorter-Symbol Trick.

Lossy compression can often create a much more compact version of data than lossless compression, but when we reverse the compression process, we do not get back exactly the original data. However, the uncompressed version is usually "good enough" for our purposes. Lossy compression relies in part on the Leave-It-Out Trick, in which we identify patterns in the data that can be simplified.

Socrative Quiz PartVI_Quiz2

IUZGAZ34E

1. A “lossy” compression scheme is a good method for temporarily compressing text files, such as a dictionary. (True, False)
2. The MP3 compression scheme for audio recordings works by replacing sounds by the corresponding musical notes. (True, False)
3. The Leave It Out Trick compresses a file by removing information that we are not likely to notice. (True, False)
4. If more than 50% of the data in an image file is removed by lossy compression, the uncompressed version will be unrecognizable. (True, False)
5. A compression scheme for movies can save a lot of space by comparing each frame to the next one, because a lot of the data will be the same. (True, False)