Laboratory 3A

Representing Numbers

Objective

- Learn how negative numbers and real numbers are encoded inside computers.

References

Software needed:

1) A web browser (Internet Explorer or Netscape)
2) Applets from the CD-ROM:
   a) Negative binary numbers
   b) Real number representations

Textbook reference: Chapter 3, pp. 52–66

Background

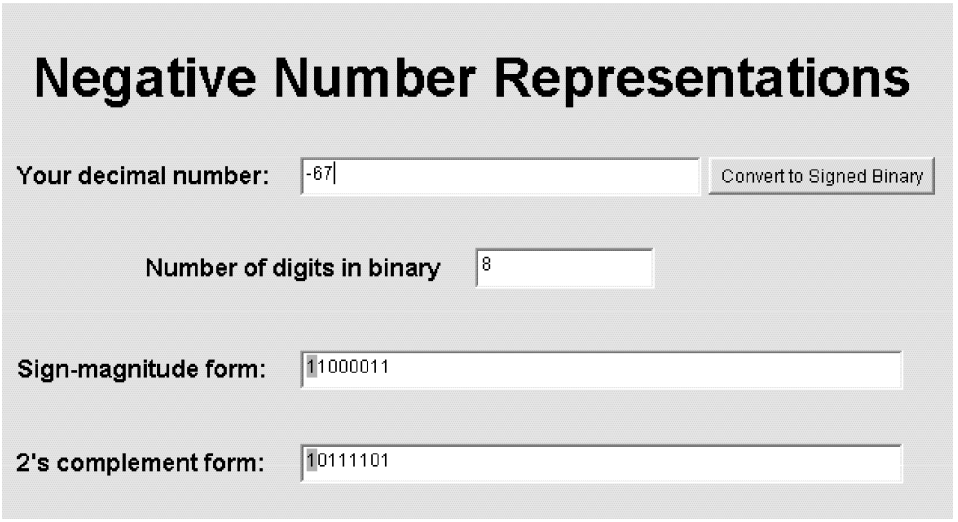
Everything you need to learn is explained in Chapter 3, "Data Representation."

Activity

Part 1

There are a number of ways to encode negative numbers in binary. Two common methods are sign-magnitude representation and two's complement representation, explained in your textbook.

To begin, start the "Negative binary numbers" applet. In the top text area, type a negative number, such as −67. Either press *Return* or click on the *Convert* button, and you should see the following:



Every representation needs to know the number of digits in the binary number, and that is set in the second text area. Eight is a common number of digits (we should be proper and call these binary digits *bits*), because a byte contains eight bits and the byte is widely used today as the smallest unit of addressable memory. (The analogy between bits and bytes and eating was too irresistible for early computer scientists, who noticed that a four-bit unit was also quite useful, especially in early coding schemes such as BCD (Binary Coded Decimal). What do you suppose they called the four-bit unit? Elementary, my dear computer science student! A *nibble*, since it is half a byte.)

The sign bit is highlighted in gray in the two bottom text areas. Try typing in a positive number and look at the forms. What sign bit do you see?

Part 2

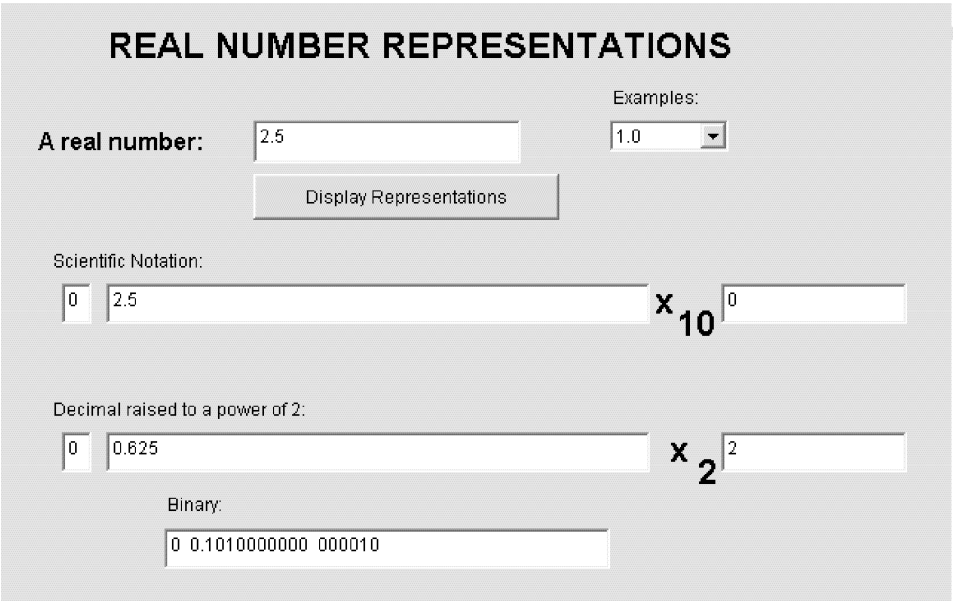
Real numbers are, well, real important to computer applications. Early computer scientists had a hard enough time figuring out the best way to represent integers. But the need for real numbers arose from the hard sciences—physics, chemistry, engineering. Computer manufacturers invented a number of ways to encode the vital pieces of a real number for science and even business. Sometimes these representation

methods were incompatible with other computers. CDC (Control Data Corporation) was one of these maverick manufacturers.

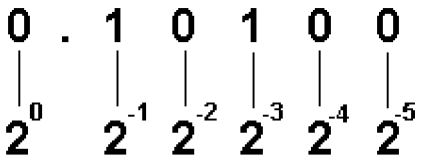
Eventually, an international standard IEEE 754 was agreed upon, and has been used in all chips since. (IEEE is the Institute of Electrical and Electronics Engineers and 754 is the number of the document defining the standard.) You can be rest assured that your computer's central processor chip uses IEEE 754 encoding.

To begin, start the "Real number representations" applet and type in some numbers with fractional parts, such as 2.5 and −373.4698. Press *Return* or click on the *Format* button. This applet re-formats the decimal real number three ways: in scientific notation (see textbook pp. 65–66), as a decimal fraction multiplied by a power of 2, and as a binary fraction.

Let's study the number 2.5. First, 2.5 is already between 0 and 10, so we merely multiply it by 10^0 (which is 1) in order to get it into scientific notation. Next, we note that the applet said $2.5 = 0.625 \times 2^2$. Since 2^2 is 4, 4 x 0.625 is 2.5. Most real number representations (including IEEE 754) place the decimal point in the leftmost position and make sure there is a 0 to the left of it. This contrasts with scientific notation, which specifies that the digit left of the decimal point must be larger than 0 but smaller than 10.



Finally, let's look at the binary representation at the bottom of the screen. 0.101 is not a decimal number but a binary one. Positional representation, which was reviewed in Chapter 2 of the textbook (p. 35), can be extended so that digits (or bits) to the right of the decimal point are multiplied by appropriate powers of 2. The first place after the 1's position (which is 20) is the halves position, or 2^{-1}. Since 2^{-1} is 0.5, we call this spot the halves place. The second place to the right of the point is the fourths place, or 2^{-2}, which is 0.25. But since there is a 0 in that place in 0.101, we do not add 0.25 to our ongoing sum. Finally, the eighths place has a 1 in it, so we add 0.125, which is 1 ÷ 8, or one-eighth.



Adding 0.5 and 0.125, we get 0.625, which is our mantissa. Multiply that by 2^2, which is 4, and we get 2.5, our original number.

Notice that a 1 appears in the two smaller text areas to the left of the numbers if the original number is negative. Real numbers use signed-magnitude representation for the overall number.

Exercise 1

Name _____ Date _____

Section _____

1) Start the "Negative binary numbers" applet.
2) Type in −1 and press *Return*. Study the result. Find the sign bit. How does the applet indicate it?
3) Change the number of bits to 32 in the text field "Number of digits in binary" and convert −1 again. Write down how the two forms of binary −1 are different in 32 bits compared to 8 bits. (Feel free to convert −1 into 8 bits again to see the different forms.)
4) Make a small table as follows:

	4	–4
8 bits		
12 bits		
16 bits		
32 bits		

5) Fill in the binary values for these two numbers, changing the number of bits from 8 to 12, then to 16, and then to 32.
6) Without using the applet, guess what −1 and −4 would look like using 64 bits. Write down your guess.

7) Change the number size back to 16 bits. Type in 32767 and press *Return*. Describe what you see below.

8) Now type in −32767 and convert. Write down the values. What is the difference between +32767 and −32767 in 2's complement form?

9) Type 4 into "number of digits in binary." Then convert the number 8. What do you see? Does this seem right?

Exercise 2

Name _____ Date _____

Section _____

1) Start the "Real number representations" applet.
2) Type in a fairly large number, like 9419876302, and press *Return*. Write down the value in scientific notation and in binary.
3) Experiment with a few other numbers. What generalization can you make about the powers of 10 and 2 that you see? (Hint: Which power is bigger? Is it always bigger by the same amount? Try dividing the bigger by the smaller. Is this ratio always the same or close to the same?)
4) How many bits does the binary representation at the bottom of the screen devote to the mantissa (the part after the binary point and separated by a space from the binary power)?

   What would be the smallest mantissa that could be represented? (Hint: You may need a calculator.)
5) Type a number such as 0.00000000123 and press *Return*. What is the sign of the exponent?
6) Type several other numbers in and press *Return*, noting the binary representation at the bottom of the screen. What observation can you make about the first bit to the right of the binary point?
7) Fill in the following table by typing each number into the top text field and pressing the "Display Representations" button.

0.5	
0.25	
0.125	

What do you notice about the mantissa (the string of bits after the point)?

What happens to the exponent?

What would the next number in the table be? (Hint: each number is 1/2 the number above it.)

Deliverables

Turn in your hand-written sheets showing your answers to the exercises.

Deeper Investigation

Computers love binary because of the bistable devices out of which we build them (discussed in Chapter 4 of your textbook, pp. 96–101). These devices have only two stable states, ideal for representing 0 and 1 but nothing else. But the fact that we like to see numbers in decimal and computers work in binary leads to some problems, similar to the impossibility of representing $^1/_3$ exactly as a decimal fraction.

In the "Real numbers representations" applet, type in 0.333 and press *Return*. Write down what you see in the scientific notation area. Then type in 0.3333 and repeat. Are you surprised? Why do you suppose the applet prints out this weird value?

Now type in 0.1 (one-tenth) and press *Return*, looking at the binary representation at the bottom of the screen. It is a bit deceptive to look at what the applet puts out. Try to come up with some combination of the negative powers of 2 that *exactly* equals 0.1. Can you do it? What does that mean about a computer representing 0.1? (You mean a computer can't even represent ten cents?!?!?!)

Because of these inaccuracies in number representations, called *roundoff errors*, early computers that were destined for business used a different method of representing dollars and cents, a method that would not be subject to roundoff errors. It was called the *packed decimal* method. See if you can find out more about the packed decimal. (One good reference is the *Encyclopedia of Computer Science*, Anthony Ralston et al., editors, Nature Publishing Group, ©2000.)

Failing that, figure out how one could use integers to represent dollar amounts exactly. What kinds of problems would still arise? (Hint: Think about interest rates!)

Laboratory 3B

Colorful Characters

Objective

- Learn how colors and text are represented.

References

Software needed:

1) A web browser (Internet Explorer or Netscape)
2) Applets from the CD-ROM:
   a) Character codes
   b) Text translator into ASCII
   c) Color maker

Textbook reference: Chapter 3, pp. 66–82

Background

Everything you need to learn is explained in Chapter 3, "Data Representation."

Activity

Part 1

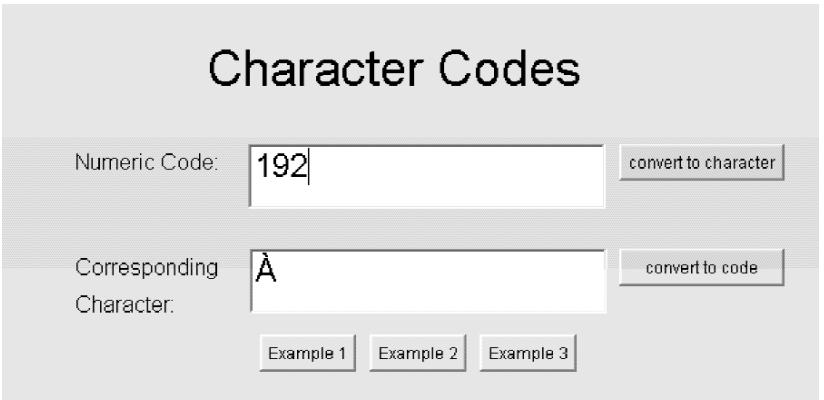
As you've learned, every kind of data in a computer must be represented by a sequence of ones and zeros. Computer scientists and engineers had to be enormously clever in representing the myriad data forms using only 1s and 0s: text, numbers, sounds, colors, still pictures, moving pictures, smells, . . . Wait! Smells? No, not yet (perhaps never!).

Now that we know how numbers of all kinds are represented using 1s and 0s, the next step is to learn how characters are represented. A character *mapping* assigns a number to each character, and then this assigned number is encoded in binary. For example, a very simple mapping scheme might assign 1 to A, 2 to B, and so on. A computer would then store B as 10, the binary representation of 2. It is as simple as that—except for making everyone agree on which mapping to use!

Pitched battles have been fought over character mappings. Well, that is a bit of an exaggeration, but it did take a long time for the computer industry to stop using many different mappings. IBM mainframes used EBCDIC while CDC (Control Data Corporation) supercomputers used their own system. DEC (Digital Equipment Corporation) and other smaller companies used ASCII. Perhaps the big breakthrough came when the IBM PC was announced in 1981. Though it was an IBM, it used ASCII and ever since ASCII has grown until it is practically the only character mapping used nowadays.

Except for Unicode, that is. With the advent of Java and web browsers and the growing international community of computer users, the Unicode character mapping has enabled us to encode virtually every written symbol as a 16-bit number. Not all software uses Unicode, and not every PC or computer can support Tibetan or Chinese script. But the mapping is in place, supported by international standards organizations. To learn more about Unicode, go to *http://www.unicode.org*.

The "Character codes" applet provides a way to type in a number, using either decimal or hexadecimal, and see the corresponding character. Following is a screenshot:



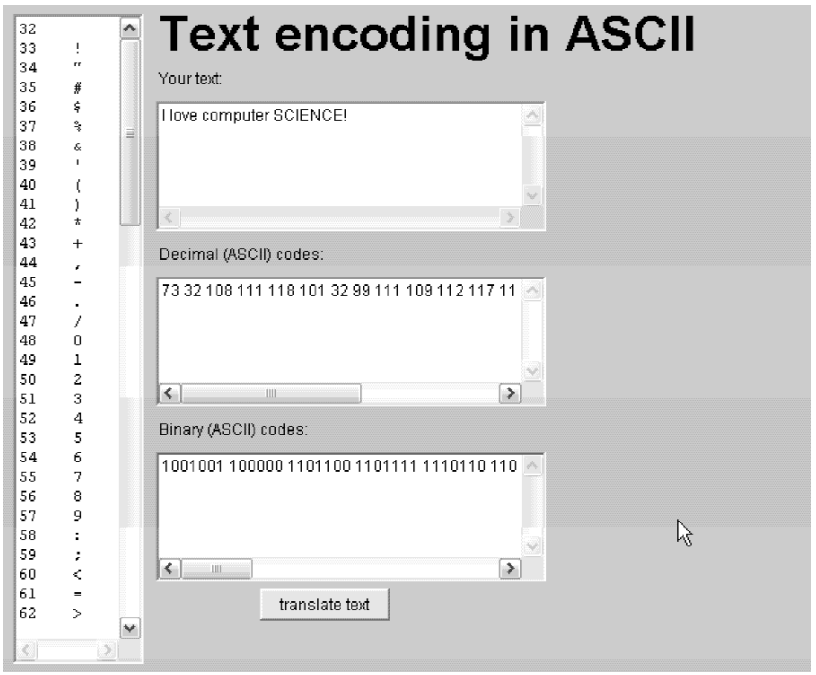
After typing in 192 in the numeric code text area, press *Return* and you will see À, which is A with a grave accent. You could also have typed `0xC0`, which is the hexadecimal number for 192. (The `0x` in front is a C-like way of designating that the following number is base 16, or hexadecimal. `x` is an abbreviation for hexadecimal.)

Try the example buttons. Example 3 cycles through all characters from 33 up to 255. If your computer can't represent a certain character, Java substitutes a little square.

Part 2

The "Text translator into ASCII" applet makes it easy to type in some text and have the computer translate all of it to binary using the ASCII character mapping.

Start the applet. Notice the character mapping in the tall text area on the left. It starts at 32, which just happens to be the blank character, and ends at 127. The codes that are mapped into the numbers 0 through 31 and 127 on up are unprintable. They have names and some are used in various functions, but in general they do not produce any graphical image on the screen.



Inside the computer's memory, there are no spaces between the seven-bit chunks that represent each character. The computer just "knows" how wide a character is and counts off bits accordingly.

Most computers today store one ASCII character in a byte, which is eight bits, not seven. The ASCII character set was extended to include 128 extra characters, many of them for PC graphics. Then Unicode came along and re-defined it all again. But the 128 codes with numeric values from 0 to 127 have remained stable.

Your textbook gives all 128 ASCII characters on p. 64. This is a two-dimensional table, unlike the one-dimensional table in the applet. For example, to find the code for "z," put together 12 and 2, which results in 122.

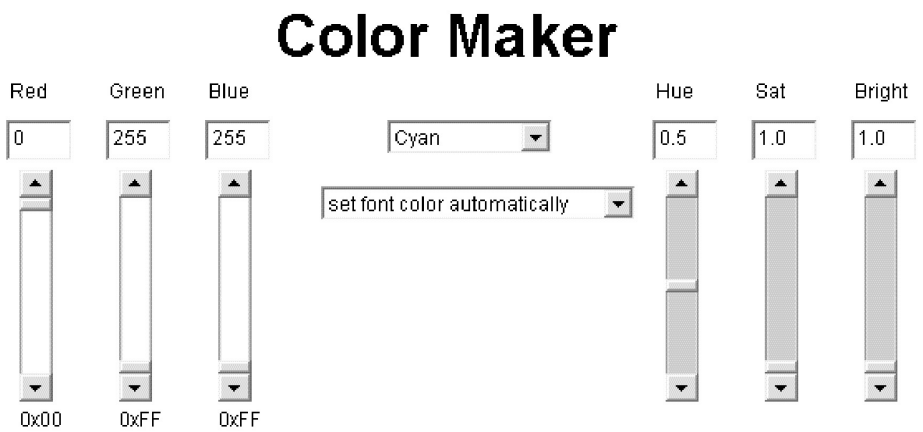
One of the characters, number 8, is called BEL. Can you guess what it does? (It sounds a beep on the computer.) Many of the others were used in early telecommunications equipment. For example, STX is "Start of text" and ACK is "Acknowledge."

Part 3

The "Color maker" applet shows you two ways that colors can be encoded using 1s and 0s. The first method is called RGB because it represents colors as combinations of three intensity values of the colors red, green, and blue. The second method is called HSB, because it represents colors as combinations of three values on the scales of hue, saturation, and brightness. RGB is discussed in detail on pp. 77–78 in your textbook.

To begin, start the "Color maker" applet, and select *Red* from the color name choice pull-down menu in the bottom left corner. The applet sets the scrollbars and text areas to the values according to the RGB encoding and the HSB encoding.

Feel free to play with the sliders and select other colors. Notice how the colors change in the bar near the bottom of the screen. The name of the color choice in the menu does not change, however, when you change the color above by means of the sliders or by typing values into the text areas.



RGB represents the red, green, and blue values as integers from 0 to 255. Since 255 is the largest number that can be represented in eight bits, it follows that RGB requires 24 bits (8+8+8) for one given color. No wonder they call it 24-bit color! Since 2^{24} is 16,777,216, there are nearly 17 million different colors that can be represented in 24 bits. There is nothing magical about using eight bits except that it matches the size of a byte, the fundamental unit of computer memory in most machines nowadays.

HSB represents the hue, saturation, and brightness as real numbers between 0 and 1.0. Leave the saturation and brightness values at 1.0 and slide the hue value up and down. Notice how the color goes from red to yellow to green to blue to magenta. The saturation number determines how much white is mixed in, and the brightness determines how strong the intensity is. If it is low, then it is like seeing the color in very dim light or a dark room.

Compare the colors *red* and *pink* by pulling down each name choice and looking at the values. In HSB, the hue value stays the same, but the saturation goes from 1.0 to about 0.31. However, the RGB version of pink shows that mixing in more green and blue with a lot of red creates pink, too. Neither RGB nor HSB is the "correct" way of viewing colors; they are just alternate ways. However, most computers store colors in RGB and convert to HSB when needed.

In summary, there are many different ways of encoding the same information in 1s and 0s. There are many different character mappings, several different ways of storing colors, and multiple ways of representing audio (see textbook p. 74).

Exercise 1

Name _____ Date _____

Section _____

1) Start the "Character codes" applet.
2) Type 75 into the top text area and press *Return* (or click on the *Convert to Character* button). Write down what you see.
3) Type in 107 and do the same. What relation does this character have to the first one?
4) Do the same for 76 and 108, and for 77 and 109. Write down the codes and the characters you see.

5) What conclusion can you draw about how upper- and lowercase characters in ASCII are represented, based on your experiments?

6) If you were writing a function to capitalize text for a word-processing program, what simple transformation would you make to the character codes to capitalize a letter?

7) Now type in 200, 201, 202, and 203, one at a time, writing down what you see.

8) What characteristics do these characters share?

9) What does ASCII code 199 look like? What language uses this character?

Exercise 2

Name _____ Date _____

Section _____

1) Start the "Text translator into ASCII" applet.
2) Type in a short sentence or phrase.
3) Press *Return* or click on the *translate* button.
4) Take a screenshot.
5) Suppose that you wrote an essay and your word processor reported that it contained 10,000 characters. How many bits would your computer need to store the essay using ASCII codes? How many bytes is that?

6) Scroll the tall window at the left to see all the ASCII codes. Write down the numerical codes for these symbols:

_____ $^1/_2$

_____ $^1/_4$

_____ $^3/_4$

_____ $\times$ (multiplication)

_____ $\div$ (division)

_____ $\pm$

Exercise 3

Name _____ Date _____

Section _____

1) Start the “Color maker” applet.
2) Select *Green* from the names choice pull-down menu. Write down the RGB and HSB values.

3) Now select *Magenta* and do the same.

4) Using more common color names, what color is magenta?

5) Suppose you heard that Kathy’s favorite color was 0,255,0. What would Kathy call that color?

6) Pick a color from the pull-down list of names in the center, and gradually slide the saturation scrollbar toward the top so that the value in the box below “Sat” goes down to 0. What color do you end up with? Is this true of any color?

7) Pick another color from the pull-down list and gradually slide the brightness scrollbar toward 0. What color do you end up with? Is this true of any color?

8) Page 78 of your textbook gives several color names that are not in the pull-down list. Let's see what *maroon* looks like. Type 140 into the Red box and press return. Then type 0 into the green box and press return and do the same for the blue box. Take a screenshot of the applet. (You may not have a color printer. If not, take a black and white screenshot anyway and ask your professor to use imagination when grading.)

Deliverables

Turn in your hand-written sheets showing your answers to the exercises. Also turn in one screenshot for the "Text translator into ASCII" applet and another screenshot for the "Color maker" applet.

Deeper Investigation

The CDC character set started with A = 1, B = 2, C = 3, and so forth. If you were creating a character mapping, this is probably what you would do. However, the CDC character set didn't have lowercase letters like a, b, or c. (The original character set used six bits and so there weren't enough possible number combinations.)

ASCII and EBCDIC do not use A=1, or even a=1. However, there are regularities in the character coding scheme. You discovered one such regularity in the previous exercises. Are there any others in ASCII that you can find?

Try to find a listing of EBCDIC's codes, most likely on the Web. You will find some surprises. For example, A=193, B=194, C=195, . . . and I=201. But J=209. What?!?!? There are no similar breaks in ASCII. There was a reason for the breaks in EBCDIC, but you may have to dig to find out what it is. (Hint: if you get stuck, punch the keys of your computer, but not too hard!)

Switching to a different train of thought, think about how arbitrary encodings tend to be. Suppose we wanted to encode the four primary directions of the compass. Would we choose North=1, South=2, East=3, and West=4? Are there any particular reasons for choosing other mappings? (Hint: Think about telling a robot to turn completely around, smoothly, without changing directions.) Also, have we shown ethnocentrism by setting North to 1? (Have you ever seen a map of the world where the South Pole is at the top? Australian restaurants love to pin these on the wall to test the sobriety of their customers.)

Write down your views on why computer scientists veer away from completely arbitrary encodings, based on all that you have learned in this lab.

Laboratory 3C

Compressing Text

Objective

- Learn how text can be compressed.

References

Software needed:

1) A web browser (Internet Explorer or Netscape)
2) Applets from the CD-ROM:
   a) Text compression using key words
   b) Text compression using Huffman encoding

Textbook reference: Chapter 3, pp. 69–74

Background

Everything you need to learn is explained in Chapter 3, "Data Representation."

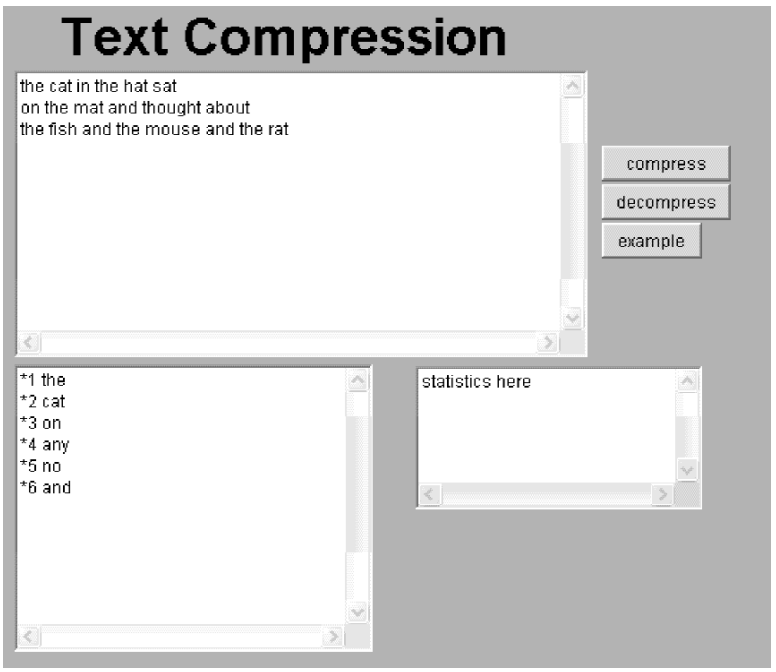
Activity

Part 1

These days when computers have lots of megabytes of RAM and vast gigabytes of disk space, you might wonder why text compression would be so important. In a word: *networking*! Today's computers are almost always networked, sending and receiving text, pictures, video, and sounds. That's a lot of data, and unfortunately the data pipes that all this stuff must go through tend to be narrow (metaphorically speaking). Lots of people are still struggling with 56 Kbps modems at home, and even T3 lines at big universities can't keep up with the demand for data. Obviously, the more we can compress data, the more data we can get through the overburdened networks!

There are many different ways of compressing a stream of 1s and 0s. Your textbook describes several of them, including run-length encoding, in which a sequence of identical characters can be replaced by a count of them. IBM used to store text files like this because there were long sequences of blanks in such files that need not be stored explicitly. Remember that a single blank is ASCII 32, so it requires eight bits like every other character.

Another method is key word encoding (textbook pp. 69–71). Start the "Text compression using key words" applet and click on the *example* button:



As you can see, the key words are listed in a table to the left, under the main text area. Each key word is identified by an asterisk followed by a number.

Click the *compress* button. The main text area is replaced with a compressed version of the text, in which the key words were replaced by their numerical codes. Decompressing should restore the text to its original poetical luster.

Notice that we only saved 8% of the total number of bits by compressing. That's not very much! Of course, our key word table is pretty small and incomplete. If we populated it with more words, we would expect the savings to go up.

Another consideration with key word encoding is that short words do not save much in compression, but long ones do. Choices have to be made, however—since it is unlikely that the word *antidisestablishmentarianism* will appear often in the newspaper or in web pages, assigning it an integer code is probably not useful.

In fact, the words *on* and *no* turn into *3 and *5, respectively, so we haven't saved any space at all. Furthermore, as we add many thousands of words to our list, the numbers themselves get rather long. It wouldn't make sense to replace *food* with *327839.

One last remark about key word encoding reveals some deeper questions about information. An old joke tells of a visitor to a comedians' convention. The visitor was surprised to see the comedians in the hall laugh uproariously as a comic onstage called out numbers: "417! 502! 29!" He finally asked the person sitting next to him what was going on.

"Well, you know," said the comedian, "we've been doing these conventions for so long that we all know the jokes by heart. To save time, we've assigned them numbers, and we just go by those."

"Well hey, even *I* can do this," thought the visitor. So he stood up and shouted "57!"

Stares and dead silence.

"I don't get it," he said to the man next to him. "Why didn't anybody laugh?"

"Sheesh," replied the comic, rolling his eyes. "Some guys just can't tell a joke."

The real joke is that without the table, explaining what the numbers mean is impossible.

Part 2

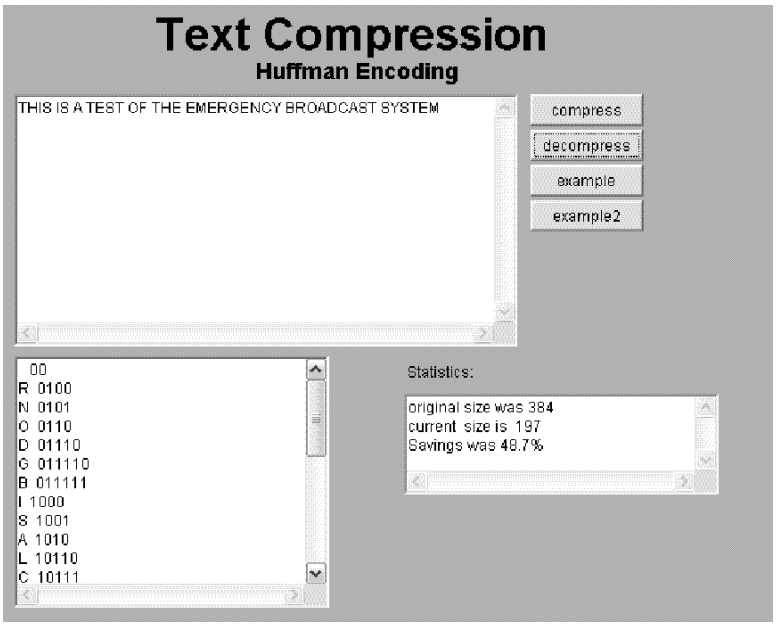
Another way to compress text is to use a variable length code, such as Huffman encoding. The key word encoding we used previously was variable length if the numbers were represented by ASCII letters, but if the numbers were stored as 32-bit integers, then it would not be variable length.

Huffman encoding is not one single mapping of letters to bit patterns. There are programs that generate the code table according to a frequency chart. Such programs are rather complicated and do not concern us here. We merely trust that they exist and can be used to create the code table.

For example, page 73 of your textbook gives a short code table that is suitable for some English words that only use the letters A, E, L, O, R, B, and D, and not having any spaces. DOORBELL is an example.

Start the "Text compression using Huffman encoding" applet and click on the *example* button. This puts DOORBELL into the text area and sets up the code table as given in the book. Click on *compress* and *decompress* to watch it work. How much room does it save?

Now click on *example 2*. This puts a longer message into the text area and sets up a table with all 26 capital letters, plus the blank. Click *compress* and *decompress* to watch it work. Notice how much it saves in space.



Huffman encoding is used quite frequently. The popular WinZip utility that compresses Windows files uses it. Such applications get even greater savings by creating a customized coding table for each document. First, the program counts the letters and makes a frequency table. Then it forms the Huffman coding table. Remember that the most frequently used letters are assigned the shortest codes. This coding table is stored along with the compressed document and must be read in before decompressing.

Exercise 1

Name _____ Date _____

Section _____

1) Start the "Text compression using keywords" applet.
2) Click the example button. Add more codes to the keyword table in the lower left corner so that the text in the top text field turns completely into asterisks followed by numbers when you click *compress*. Take a screenshot.
3) Could this compression method be used for encryption as well? (Encryption is when you scramble the message so that hostile eavesdroppers cannot make sense of it.) State why you believe it could or couldn't be used for encryption.

4) Enter a word preceded by an asterisk into the text area and press *compress*. What does the applet do? Does it decompress this correctly?

5) What would happen if you really needed to have the sequence *3 in your text? How would you modify the applet so it wouldn't get confused? (Hint: Choose another numbered code, like *99, and let that represent *5.)

Exercise 2

Name _____ Date _____

Section _____

1) Start the "Text compression using Huffman" encoding applet.
2) Click on *example 2* and *compress*.
3) Remove the first bit from the beginning of the encoded message and click *decompress*. What happens?

4) Take a screenshot.
5) What generalization can you make about Huffman encoding versus run-length or keyword encoding? How resilient are these methods to errors, which would probably arise as the bits flow through a network?

6) Clear the text area and type in XXXXXXXX, and click *compress*.
7) What happens to the space savings? Are you surprised?

8) The sequence of 1s and 0s used to represent a particular letter in a Huffman code is not arbitrary or random. Your textbook on p. 73 hints at the way a sequence is chosen. Given this, what can you say about the letter J? What about the letter E?

Deliverables

Turn in your hand-written sheets showing your answers to the previous exercises. Also turn in one screenshot for the "Text compression using key words" applet and another screenshot for the "Text compression using Huffman encoding" applet.

Deeper Investigation

Now is a great time to discuss one of the classic questions of computer science. There seems to be a trade-off between time and space, between computing effort (measured in seconds) and memory requirements (measured in memory cells). A program that goes to extra effort to produce the very smallest compressed file takes longer to run than a program that uses a pre-assigned coding table.

Here is the classic question: Are time and space always *inversely proportional*? (This means when one goes up, the other goes down.) In other words, as you save on time do you use more space? But if you spend more time computing, can you use less space? This trade-off is seen in many areas of computer science, but nobody can prove that it is truly necessary. (This "time versus space" question is really at the heart of the question as to whether Algorithm Class P is the same as Algorithm Class NP, discussed in Chapter 17 of your textbook, pp. 557–558.)

Think of some aspects of daily life in which you must consider trade-offs. It doesn't have to be time versus space, but it can be. For example, consider the following trade-off that comes to mind: The most difficult and time-consuming degrees yield the best-paying jobs. Do you agree?