7

* Data Representation

The projects in this chapter explore ideas of how data is represented inside a computer. We first look at straightforward methods for representing things like numbers and characters as binary numbers. We then look at two ideas related to data representation: text compression, which considers how to pack the same amount of information into a smaller space, and error detection, which is concerned with examining the representation of a piece of data to discover whether the data has changed since it was first created.

7.1 Fixed Length Codes

One of the fundamental relationships in computer science is that there are \(2^n\) different \(n\)-bit binary numbers, i.e. \(2^n\) patterns of 1s and 0s that are \(n\) characters long. What this implies is that if we have a set of items and we want to define a different binary pattern for each item the number of bits in each pattern must be at least

\[
k = \lceil \log_2 m \rceil
\]

where the notation \(\lceil x \rceil\) means “the smallest integer greater than \(x\)” (known as the “ceiling” of \(x\)).

Tutorial Project

The first set of exercises use IRB to explore the relationship between the pattern length and number of patterns.

T1. The `log` method in Ruby computes the natural logarithm (\(\log_e\) or \(\ln\)). For example,

```ruby
>> Math.log(10)
=> 2.302585092994046
```

T2. Ruby does not have a function to compute \(\log_2\) but it’s easy to define one, using the fact that

\[
\log_2 x = \frac{\log_e x}{\log_e 2}
\]

Type this expression to define a new method named `log2`:

```ruby
>> def log2(x) Math.log(x) / Math.log(2.0) end
```
(or, if you want, download the file log2.rb from the class web site and load it into IRB).

T3. Try out your new log2 method on a few different values:

```ruby
>> log2(8)
=> 3.0
>> log2(10)
=> 3.2192809488736
>> log2(16)
=> 4.0
```

Are these the correct results? If you have a calculator with a log₂ function you can evaluate these same expressions with your calculator; if not, try computing 2ˣ for these values, e.g. see if 2³⁺² ≈ 10.

T4. Ruby has a method named ceil (for “ceiling”) that “rounds up” a floating point number to the next higher integer. Try calling it on the results created by the log2 method:

```ruby
>> log2(20)
=> 4.32192809488736
>> log2(20).ceil
=> 5
>> log2(100)
=> 6.64385618977473
>> log2(100).ceil
=> 7
```

Can you see from these results that if we have a set of 20 items it will take 5 bits to make a unique pattern for each one, or if we have 100 items it will take 7 bits?

### 7.2 Data Compression and Huffman Codes

Download the program in the file named huffman.rb and the letter frequency data in aafreq.txt.

**Tutorial Project**

The key to the algorithm that compresses text is a data structure known as a *priority queue*, which is basically just an array that is always sorted.

T5. Load the program into IRB:

```ruby
>> load "huffman.rb"
=> true
```

T6. Make a new priority queue object and save it in a variable named pq:

```
>> pq = PriorityQueue.new
=> []
```

As you can see from the result above the new queue is empty.

T7. Priority queues, like arrays in Ruby, can contain any type of object, as long as the objects can be compared (if they can't be compared to one another there is no way to sort them). Try adding a couple of strings to your new queue:

```
>> pq.insert("lemon")
=> ["lemon"]
>> pq.insert("grape")
=> ["grape", "lemon"]
```
7.2 Data Compression and Huffman Codes

Note that the only argument passed to the `insert` method in the calls shown above is the item to add to the queue. The queue figures out where to insert the object and automatically puts it in a place that makes sure the queue remains sorted.

T8. Add a few more strings to your queue:

\[
\text{\texttt{\{kiwi, strawberry, pear\}.each \{ |x| pq.insert(x) \}}}
\]

\[
\Rightarrow \{\text{"kiwi", "strawberry", "pear"}\}
\]

T9. Check to make sure the queue now has 5 items, and that they are stored in the queue in alphabetical order:

\[
\text{\texttt{pq.length}}
\]

\[
\Rightarrow 5
\]

\[
\text{\texttt{pq}}
\]

\[
\Rightarrow \{\text{"grape", "kiwi", "lemon", "pear", "strawberry"}\}
\]

T10. Even though the queue looks like an array, it is not an array. Like any object, you can ask Ruby to tell you which class it belongs to:

\[
\text{\texttt{pq.class}}
\]

\[
\Rightarrow \text{PriorityQueue}
\]

T11. If you try to call methods that work for arrays you will get an error message:

\[
\text{\texttt{pq << "orange"}}
\]

\[
\text{NoMethodError: undefined method \'<<\' for ...}
\]

\[
\text{\texttt{pq.sort}}
\]

\[
\text{NoMethodError: undefined method \'sort\' for ...}
\]

This makes sense, because if the class allowed these other operations the order of the items in the queue might change, and the items would no longer be stored according to their priority.

T12. When items are removed from the queue they should be taken from the front. The `shift` method does exactly that. Type these expressions to remove the first two strings from your queue and then make sure there are still three items left in the queue:

\[
\text{\texttt{s = pq.shift}}
\]

\[
\Rightarrow \text{"grape"}
\]

\[
\text{\texttt{t = pq.shift}}
\]

\[
\Rightarrow \text{"kiwi"}
\]

\[
\text{\texttt{pq}}
\]

\[
\Rightarrow \{\text{"lemon", "pear", "strawberry"}\}
\]

The items stored in the priority queue in the Huffman tree algorithm are tree nodes. In the Ruby program these items are objects that belong to a class named `Node`.

T13. To make a new leaf node (a node that corresponds to a single letter) call `Node.new` and pass it a letter and its frequency:

\[
\text{\texttt{t0 = Node.new(\"A\", 0.2)}}
\]

\[
\Rightarrow ( A: 0.200 )
\]

\[
\text{\texttt{t1 = Node.new(\"B\", 0.3)}}
\]

\[
\Rightarrow ( B: 0.300 )
\]

The parentheses in the output strings shown above are intended to remind us that the values are tree nodes – think of these as parts of the circle that would surround the letter and its frequency if we were drawing a picture.

T14. A method named `combine` will create a new interior node by combining two existing nodes:

\[
\text{\texttt{t2 = Node.combine(t0, t1)}}
\]

\[
\Rightarrow ( 0.500 ( A: 0.200 ) ( B: 0.300 ) )
\]

© 2008 John S. Conery. Please do not distribute without permission.
Figure 7.1: Huffman tree for the 20 letters used in amino acid strings.

The string printed by Ruby for an interior node is slightly more complicated, but with practice you will recognize the parts. The outer set of parentheses show that this is a node. The 0.5 after the open parenthesis means the combined node has a frequency of 0.5, which is the sum of the frequencies of the two nodes passed as parameters. The two descendants of this interior node are shown following the frequency.

The next set of exercises will use the letter frequency data in the file aafreq.txt to make a Huffman tree for the 20 letters used to describe protein sequences.

T15. Use the method named readFrequencies to read the data and make an associative array (aka a "hash") to hold the data:

```ruby
>> freq = Huffman.readFrequencies("aafreq.txt")
=> {"V"=>0.06, "K"=>0.059, ... }
```

T16. To see the frequency of one of the letters, just access that item in the array:

```ruby
>> freq["L"]
=> 0.094
>> freq["W"]
=> 0.011
```

Are these the frequency values shown in the table in the lecture slides?

T17. A method named initQueue will make a priority queue, then make a Node object for each item in the hash and add the Node to a queue:

```ruby
>> pq = Huffman.initQueue(freq)
=> [( W: 0.011 ), ( C: 0.021 ), ... ( S: 0.083 ), ( L: 0.094 )]
```

Does this look accurate to you? Are the items in the queue Node objects? Are they sorted? How many are there?

T18. To build a Huffman tree we will execute a loop that removes two nodes, combines them, and inserts the new node back into the queue. These three statements illustrate how it's done:
7.2 Data Compression and Huffman Codes

```ruby
>> n1 = pq.shift
=> (W: 0.011)
>> n2 = pq.shift
=> (C: 0.021)
>> pq.insert(Node.combine(n1, n2))
=> [(M: 0.023), (H: 0.025), ...]
```

T19. As you can see, the first two nodes (for C and W) have been removed, and the front of the queue is now the node for M. Can you find the new node in the middle of the queue? Is it where it belongs?

T20. How long is the queue after executing the three statements shown above?

T21. You can do all three of these operations in a single Ruby expression:

```ruby
>> pq.insert(Node.combine(pq.shift, pq.shift))
=> [(Y: 0.031), (0.032 (W: 0.011) (C: 0.021), ...]
```

T22. If you want to continue to make the tree, to see how the algorithm works, just re-execute the statement from the previous problem several more times. How many times will you have to execute the command to make the complete tree?

Note: it's not easy to see the tree structure when the nodes are just printed on the terminal, like they are in the previous examples. I'm working on a way to have Ruby generate pictures as we type expressions so we can watch the tree grow.

T23. A method named `buildTree` automates the steps in the previous problem – it initializes a priority queue with nodes for each letter in a hash and then repeats the remove-and-combine step until the queue has been reduced to a single node. That node is the root of the tree:

```ruby
>> tree = Huffman.buildTree(freq)
=> (1.000 (0.420 (0.193 ...)))
```

T24. A method named `encode` will create a pattern of bits for a string. Note that we have to pass both the string to encode and the tree that defines the encoding as parameters when we call `encode`:

```ruby
>> Huffman.encode("W", tree)
=> "101010"
>> Huffman.encode("L", tree)
=> "1111"
```

Do these results look right?

T25. You can pass any length string you want to encode:

```ruby
>> Huffman.encode("MLSVV", tree)
=> "001001111111010001000"
```

T26. If you include the :`trace` option with the call the method prints out the letters as it encodes the string, to help you make sure the correct encoding is produced:

```ruby
>> Huffman.encode("MLSVV", tree, :trace)
M => 00100
L => 1111
S => 1110
...```

T27. Save the encoding of a test string so we can use it for some later tests:

```ruby
>> bits = Huffman.encode("MLSVV", tree)
=> "001001111111010001000"
```

T28. The method that converts 1s and 0s into letters is named `decode`. Pass this method the string to decode and the tree that defines the code:
7 * Data Representation

>> Huffman.decode("1111", tree)
=> "L"
>> Huffman.decode("0000", tree)
=> "Q"

T29. Let's see if the string created by encode can be decoded properly:

>> Huffman.decode(bits, tree)
=> "MLSVV"

Did you get back the string you originally passed to encode?

T30. You can also pass the :trace option to decode:

>> Huffman.decode(bits, tree, :trace)
0 00100 M
5 1111 L
9 1110 S
...

The number at the front of each line is the location where each letter was found, e.g. the letter M was found at location 0, the letter L starting at location 5, and so on.

7.3 Error Correction and Hamming Codes

The goal of a text compression algorithm is to reduce the number of bits required to store some information in memory or in a file. The next set of projects goes in the opposite direction – we’re going to add bits to the representation of a piece of text in order to help detect errors that might occur when the text is stored or transmitted.

Tutorial Project

These exercises use new methods defined in the file parity.rb. The method hex makes a string of hexadecimal (base 16) digits for a number, and binary makes a string of binary digits.

T31. Start a new Ruby session and load the methods in parity.rb:

>> load "parity.rb"
=> true

T32. Use the hex method to get the hexadecimal strings for a few numbers:

>> hex(12)
=> "C"
>> hex(23)
=> "17"
>> hex(255)
=> "FF"

T33. Repeat the above, but this time get the binary strings:

>> binary(12)
=> "1100"
>> binary(23)
=> "10111"
>> binary(255)
=> "11111111"
7.3 Error Correction and Hamming Codes

T34. The hex and binary methods allow us to pass an extra parameter that controls how many digits to include in the string. To make an 8-bit binary number pass the number 8 as a second parameter:

```ruby
>> binary(12, 8)
=> "00001100"
```

Note that this binary number is the same as the one created by the call to `binary(12)`; it has just been “padded” with leading 0s to make the result 8 digits long.

T35. Create a string to use for experimenting with parity bits:

```ruby
>> s = "ABCDEF"
```

T36. An expression of the form `s[i]` returns the ASCII code for the character in location `i` of the string:

```ruby
>> s[0]
=> 65
>> s[1]
=> 66
```

Verify these numbers are correct by looking up the ASCII codes (either by searching for ASCII at Wikipedia or using the tables shown in the lecture notes).

T37. The `each_byte` method is an iterator that will execute a statement for each character in a string:

```ruby
>> s.each_byte { |x| puts x }
65
66
67
...
```

T38. Use the `binary` method to print the 8-bit binary version of the ASCII codes:

```ruby
>> s.each_byte { |n| puts binary(n, 8) }
01000001
01000010
01000011
...
```

Do these results correspond to the values you found in the ASCII tables from Wikipedia?

T39. Try modifying the previous expression so in prints 2-digit hexadecimal codes instead of 8-digit binary. Do these strings correspond to the entries in the ASCII table?

The term *parity* refers to the number of 1 bits in a binary pattern. If there are an even number of 1s we say the pattern has even parity. For example, the string “0101” has even parity, but “0111” has odd parity because it has three 1s.

The simplest form of error detection is to add an extra bit, called a *parity bit*, to a string before saving it or transmitting it. The idea is to make the new string have even parity. If a single bit changes during the time the string is in transit the receiver will read a string that has odd parity.

T40. A method named `parity_bit` will figure out which binary digit to add to a binary string in order to give it even parity. Try calling it on a couple of short strings:

```ruby
>> parity_bit("010")
=> "1"
>> parity_bit("0101")
=> "0"
```

Note that `parity_bit` returns “1” when the input string has an odd number of 1s and returns “0” if the input has an even number of 1s. Try calling this method several more times with
your own strings of 0s and 1s to make sure you understand what this `parity_bit` method does.

T41. The lecture slides showed examples of parity bits for ASCII characters. We can use the `parity_bit` method to figure out the parity of an ASCII code by first creating the binary pattern for a code and then passing it to `parity_bit`:

```ruby
>> s[0]
=> 65
>> binary(s[0], 7)
=> "1000001"
>> parity_bit(binary(s[0], 7))
=> "0"
```

Since ASCII characters are 7 bits long we should pass the number 7 in calls to `binary` so we always get back 7-letter binary strings.

T42. A method named `add_parity` will attach a parity bit to a character:

```ruby
>> add_parity_bit(binary(s[0], 7))
=> "1000010"
```

Note that the string returned by the call to `binary(s[0], 7)` is 7 letters long, and adding a parity bit made an 8-digit string.

T43. If you want to see how this works for all the letters in the test string `s`:

```ruby
>> s.each_byte { |ch| puts add_parity_bit(binary(ch, 7)) }
10000010
10000100
10000111
...
```

Can you see how each of these strings corresponds to the ASCII code for a letter with a parity bit attached to the end? Do each of the strings produced by the call to `add_parity_bit` have even parity?

T44. Make a more interesting string to use for the next set of tests:

```ruby
>> s = "Hello, Ruby."
=> "Hello, Ruby."
```

T45. A message is an array of 8-bit patterns, where each pattern is formed by attaching a parity bit to the 7-bit ASCII code to letters in a string. Pass a string to a method named `make_message` to make a message from the characters in that string:

```ruby
>> m = make_message(s)
=> ["10010000", "11001010", "11011000", ...]
```

Make sure you understand where each of these patterns came from: the first is the ASCII code for the letter “H” with its parity bit tacked on to the end, the next is for the letter “e,” and so on. Do each of these letters have the correct parity?

T46. To test the error-detecting ability of a single parity bit, a method named `garble` will add random errors to a message. A call of the form `garble(m, n)` will add `n` errors to the message `m`. Try it out by adding three errors to the message you just made:

```ruby
>> garble(m, 3)
=> ["10010000", "11001010", "11011000"
```

Can you spot the errors?

T47. Call a method named `check_message` to look for errors in a message. This method will print an error message if it finds a parity error, and then return the string made by the letters in the message. Try it out on your message:
7.3 Error Correction and Hamming Codes

>> check_message(m)
found parity error in character 5: 01011000
found parity error in character 6: 01001001
found parity error in character 9: 11001101
=> "Hello,$Ruby."

Since the errors were inserted into random locations in the message your output won’t look like the string above.

T48. From the error messages printed by check_message can you see where the bits were changed?

T49. According to the output of check_message, there were three errors in m. But the string returned by the method has only two differences from s: the space was turned into a dollar sign, and the “b” in “Ruby” was turned into an “f.” Where do you think the third error occurred? Did something like this happen with your own example? Or can you find all the letters that changed from your original string s to the string returned by check_message?

T50. Add a lot more errors to the message, and call check_message again:

>> garble(m, 10)
=> ["11010010", "11001011", "11011000", ...]
>> check_message(m)
found parity error in character 1: 11001011
found parity error in character 4: 11011010
found parity error in character 7: 10100111
found parity error in character 8: 10101011
found parity error in character 11: 01010100
=> "ielNm, SybYs"

There should now be around 13 single-bit differences between the original message and the current contents of m. The output from check_message is certainly scrambled up pretty well, and it looks like there are 13 errors here. But check_message printed only five error messages. Can you explain why?