Binary representations for numbers

Help Center

In this activity, we will describe several simple recursive algorithms for generating binary representations of non-negative integers and converting these representations back into decimal form. These methods provide simple illustrations of the power of recursive algorithms.

Binary numbers

A decimal number is typically modeled as a sequence of the decimal digits (0 to 9).) A binary number is expressed as a sequence of binary digits (either 0 or 1) often called *bits*. Binary numbers are particularly important in computer science since they are the fundamental representation used to model the digital logic that underlies all modern computer hardware.

Creating a list of all binary numbers of a given length is not particularly difficult. For example, the iterative method in `gen_all_sequences` can enumerate all 2^n binary numbers of length n. Your first task for this activity is to write a recursive function `make_binary(length)` that returns a list containing all binary numbers of the specified length.

Observe that this problem has a natural recursive structure. If n is zero, your implementation should return a list consisting of exactly one string, the empty string `""`. Otherwise, your implementation should compute `make_binary(length - 1)` and use the resulting list to construct the final list. Once you have attempted the problem, you are welcome to examine our code which contains solutions to all four of the problems in this exercise.

Computing the value of a binary number

For decimal numbers, the leftmost digits are usually treated as being more significant that the rightmost digits. This convention is also followed in binary numbers. If the binary number has the n bits $b_{n-1}b_{n-2}\ldots b_1b_0$, then the corresponding decimal value of this number is $\sum_{i=0}^{n-1} 2^i b_i$. For example, the binary string `"100"` has the decimal value 4 while the binary string `"001"` has the decimal value 1. As a point of interest, notice that our solution for `make_binary` generates the binary strings ordered by ascending value.

Your next task is to write a recursive function `bin_to_dec(bin_num)` that computes the decimal value of the specified binary number. While the iterative definition given above can be used to compute this value, we suggest that you implement `bin_to_dec` recursively to gain further practice with recursion. If the length of the binary string is zero, your implementation should return zero. Otherwise, your implementation should compute the decimal value of the $n-1$ most significant bits and then use this value along with the value of the least significant bit to compute the decimal value of all n bits. Once you have attempted the problem, you are welcome to examine our sample solution.

Gray codes

The standard binary representation, while incredibly useful in many applications, is not always the best binary representation for some applications. For example, imagine an application in which one is scanning an n-bit number and the scanner occasionally misreads one of the bits. In the standard binary representation, if the misread bit is a very significant bit, the corresponding value of the binary string may differ substantially from the intended value. For example, the five-bit binary strings

`"00101"` and `"01101"` differ in only one bit but represent the numbers 5 and 13, respectively.

Gray coding (also known as reflected binary numbers) is an alternative numbering of binary strings in which strings corresponding to consecutive values always differ by exactly one bit. In the application above, if the scanner misreads at most one bit, the value of the scanned Gray code is at most one away from the correct value. In many cases, bounded errors of this type are perfectly acceptable.

For the third problem, your task is to write a recursive function `make_gray(length)` that generates a list of binary strings of the specified length *ordered such that consecutive strings differ by exactly one bit*. Your recursive solution should be very similar to `make_binary` which generated a list of all binary strings ordered by their standard values. If the length is zero, the answer is the list consisting of the empty string. Otherwise, your function should recursively compute `make_gray(length-1)` and use this list to construct `make_gray(length)`.

In the case of standard binary numbers, `make_binary(length)` created two copies of the list `make_binary(length - 1)`. `make_gray(length)` should also create two copies of `make_gray(length - 1)`. However, one of these copies should be reflected (i.e; reversed). Spend a few minutes experimenting with some simple examples and see if you can implement `make_gray`. If you need more help, this section of the Wikipedia page on Gray codes explains the solution.

Computing the value of a Gray code

Our solution for `make_gray` returns a list of Gray codes ordered in ascending value. For standard binary numbers, the function `bin_to_dec` computes the value of a binary number. To compute the decimal value of a Gray code, we will implement a function `gray_to_bin` that converts a Gray code to the standard binary number with same value. This function can then be used to compute the value of a Gray code by evaluating `bin_to_dec(gray_to_bin(gray_code))`.

Challenge problem: Your final task is to write a recursive function `gray_to_bin(gray_code)` that performs this conversion. Again, this function is remarkably simple. However, deriving this function on your own may prove to be quite difficult. So, we recommend that you read up further on Gray codes as you work on this problem. In particular, we suggest that you focus on the next to last paragraph in the Wikipedia section referenced above.

Created Mon 30 Jun 20149:49 PM BST

Last Modified Sat 19 Jul 2014 5:04 AM BST