

MAT 3670: Lab 3

Bits, Data Types, and Operations

Background

In previous labs, we have used the Turing machine to manipulate bit strings. In this lab, we will continue to focus on bit strings, placing more emphasis on the interpretation of these strings, as explained in Chapter 2 of our text.

Although Turing machines could be used for this week's lab exercises, you would probably agree that the design of these machines would be somewhat complicated, even though the algorithms themselves are fairly basic. To make life simpler, we will use a different way to implement these algorithms.

All of the algorithms for this lab are to be implemented in the **Python**[†] language. Our goal is not to learn the Python language, per se, but rather to have an environment in which it is relatively easy to experiment with various data types and operations on bit strings. Python is an interpreted language and shares many ideas from languages such as Java and C++.

Since Python is an interpreted language, it encourages experimentation: the interpreter immediately acts upon what you type. Some class time will be devoted to Python, but mostly you will be expected to find out what you need for this lab by consulting web resources and imitating examples you see in class. Since you already know at least one high-level language (probably Java or C++), you won't find this to be too difficult.

Official documentation for Python can be found at <http://docs.python.org/>. Once there, click on the **Tutorial** link for introductory information about the language. You can use this website to answer questions you may have about Python.

Pre-Lab Exercises

During lab, you will be asked to implement the functions described in Figure 1. To prepare yourself for this activity, do the following before this week's lab:

- Read and understand the functions described in Figure 1.
- Provide pseudo-code for each of the functions of Figure 1.
- Review all code (begins on page 5) found in `lab3.py`.
- As time allows, create Python implementations based on your pseudo-code.

Laboratory Exercises

1. Descend to your 3670 directory; create a **lab3** directory to store the files you will create for this lab. Download `lab3.py` from the course website and place it in your **lab3** directory.
2. Open `lab3.py` with Aquamacs. (This is probably the default application which opens `.py` files, but to be sure, right click on `lab3.py` and verify the **Open With** default application.)

[†]Python is open source and is free to use. If you want to run Python on your own computer, see www.python.org for download information. Depending on your computer and operating system, it may already be pre-installed.

- **successor(A)**

This function is given a list of bits, A , and is to determine its binary successor. For example, the successor of $[1, 0, 1, 0, 1, 1]$ is $[1, 0, 1, 1, 0, 0]$. This is the same action carried out by one of your Turing machines from Lab 2. Your algorithm should find the successor using the same technique described in that lab; i.e., search for the rightmost 0, complement this bit and all remaining 1 bits to its right.
- **complement(A)**

This function is given a list of bits, A , and is to compute its bit-wise complement. For example, if $A = [1, 0, 1, 0, 1, 1]$, its complement is $[0, 1, 0, 1, 0, 0]$.
- **twosComplement(A)**

This function is given a list of bits, A , and is to compute its two's complement. For example, if $A = [1, 0, 1, 0, 1, 1]$, its two's complement is $[0, 1, 0, 1, 0, 1]$.
- **add(A, B)**

Given two lists of bits, each of the same length, this function should perform binary addition and return the sum. The carry-out from the most significant bit is to be discarded. For example, if $A = [0, 0, 1, 1]$ and $B = [0, 0, 1, 1]$ the sum is $[0, 1, 1, 0]$.
- **subtract(A, B)**

Given two lists of bits, each of the same length, this function should perform the subtraction $A - B$. Hint: using properties of the two's complement number system, this can be done using several of the functions above.
- **decimalToUnsigned(n, m)**

Given a non-negative integer value n , this function computes the list of bits which encodes this value, as an unsigned, binary value. The second parameter specifies how many bits should appear in the result. For example, if $n = 12$ and $m = 6$, the desired result is $[0, 0, 1, 1, 0, 0]$. If the specified value m is too small, then the fewest number of bits possible should be used.
- **signedToDecimal(A)**

Given a list of bits A which represents an integer value in the two's complement system, this function determines this value. For example, if $A = [1, 0, 0, 1]$, the value to be determined by this function is -7 .

Figure 1: Functions to be implemented for Lab 3.

3. Launch the Python interpreter from within Aquamacs by selecting the **Python** menu, then choosing **Start Interpreter**. In response, the window will be subdivided into two parts: the editor window and a new interpreter window. In the interpreter window, you can interact with Python by entering statements at the `>>>` prompt.

Try each of the following:

```
A = [1, 0, 0, 1, 1]
len(A)
range(10)
range(len(A))
A[0]
A[1]
A.append(1)
A
A.reverse()
```

4. To make the interpreter aware of the definitions stored in `lab3.py`, enter the following at the `>>>` prompt:

```
import lab3
```

Afterwards, you can access function definitions within `lab3.py`, such as:

```
lab3.bitPopulation(A)
```

Read the definition of this function, found in `lab3.py`, to see how it works.

5. Each function definition within `lab3.py` begins with a “documentation string”, set off with three double quotes. To see how they are used, enter this at the `>>>` prompt:

```
print lab3.bitPopulation.__doc__
```

Other comments within the function are indicated with `#`. **All code you supply for this and other labs should be generously commented.**

6. Modify `bitPopulation` so it counts the total number of 0’s in a given bit sequence. After editing the file `lab3.py`, be sure to save the file. Also, you need to inform the interpreter that the definitions have changed by entering:

```
reload(lab3)
```

at the `>>>` prompt. *Each subsequent change to `lab3.py` must be followed by a request to reload the file.*

7. For the remainder of the lab, your task is to complete the functions from the pre-lab exercises, placing their definitions in `lab3.py`. Notice that each of these functions appears as a stub within `lab3.py`.

These functions can be completed and tested one at a time. After you have convinced yourself that your definition is correct, use the appropriate “demo” function to give a more thorough workout of your functions. These functions are summarized in Figure 2.

8. To gracefully exit from your Python session, enter the control-d key combination at the `>>>` prompt. You can then close your Aquamacs session.

Submissions

When you have completed the lab, submit your `lab3` folder by dragging it onto the EIU submission icon.

<code>successorDemo</code>	Cycles through the 2^n bit patterns of n -bit quantities
<code>complementDemo</code>	Displays bit patterns and their complements
<code>twosComplementDemo</code>	Displays bit patterns and their two's complement values
<code>addDemo</code>	Displays the result of adding all possible pairs of n -bit binary quantities
<code>subtractDemo</code>	Like <code>addDemo</code> , but for subtraction
<code>numberDemo</code>	Displays all bit patterns of n bits each and their numerical representation in the unsigned and signed, two's complement systems.

Figure 2: Demo functions for Lab 3. *Warning:* many of these functions produce potentially large amounts of output, so it is important to keep their arguments relatively small. For example, choosing n to be 3 or 4 will limit the output to a manageable amount, yet still provide informative output.

Contents of lab3.py

```
1  """
2  Python functions for MAT 3670, lab 3.
3  """
4
5  def bitPopulation(A):
6      """
7      bitPopulation(A) reports the number of bits in A which are set to one.
8      """
9
10     # Establish a counter
11     count = 0
12
13     # Iterate over all bits of A
14     for i in range(len(A)):
15         if A[i] == 1:
16             # Another one bit has been seen, so we count it
17             count = count + 1
18
19     # Hand back the result
20     return count
21
22
23 def successor(A):
24     """
25     successor(A) returns the binary successor of a given bit string A.
26     """
27     # STUB: TO BE COMPLETED
28
29
30 def complement(A):
31     """
32     complement(A) returns the bit string obtained by complementing each bit of A.
33     """
34     # STUB: TO BE COMPLETED
35
36
37 def twosComplement(A):
38     """
39     twosComplement(A) computes the two's complement of a given bit string A
40     """
41     # STUB: TO BE COMPLETED
42
43
44 def decimalToUnsigned(n, m):
45     """
46     decimalToUnsigned(n, m) produces a list of the m bits corresponding
47     to the decimal value n, assumed to be non-negative. If m is too small,
48     then the fewest possible number of bits is used. The resulting list
49     of bits is treated as an unsigned value.
50     """
51     # STUB: TO BE COMPLETED
52
53
54 def signedToDecimal(A):
55     """
```

```
56     signedToDecimal(A) yields the numerical value corresponding to A,
57     a list of binary digits, represented in two's complement form.
58     """
59     # STUB: TO BE COMPLETED
60
61
62     def unsignedToDecimal(A):
63         """
64         unsignedToDecimal(A) yields the numerical value corresponding to A,
65         a list of binary digits, represented as an unsigned value.
66         """
67
68         # Process the bits in A from left to right, from most significant to least significant
69         value = 0
70         for i in range(len(A)):
71             value = 2*value + A[i]
72
73         return value
74
75
76     def add(A, B):
77         """
78         Performs binary addition on the bit strings A and B, yielding
79         the sum. Assumes both arguments have the same number of bits.
80         """
81         # STUB: TO BE COMPLETED
82
83
84     def subtract(A, B):
85         """
86         subtract(A, B) performs the subtraction A - B. Assumes both arguments have the
87         same number of bits.
88         """
89         # STUB: TO BE COMPLETED
90
91
92     def generateAllBitPatterns(n):
93         """
94         generateAllBitPattern(n) returns a list of all possible bit patterns of n bits each.
95         """
96
97         return [decimalToUnsigned(i, n) for i in range(2**n)]
98
99
100    def successorDemo(n):
101        """
102        Uses the successor function to cycle through all possible bit patterns of n bits
103        """
104
105        # start with the pattern of n 0's
106        p = n * [0]
107
108        # cycle through all bit patterns
109        for i in range(1 + 2**n):
110            print p
111            p = successor(p)
112
```

```
113
114 def complementDemo(n):
115     """
116     complementDemo(n) outputs the complement of each bit pattern of n bits.
117     """
118
119     for p in generateAllBitPatterns(n):
120         # get the complement of p and display the result
121         pc = complement(p)
122         print p, pc
123
124
125 def twosComplementDemo(n):
126     """
127     twosComplementDemo(n) demonstrates the property that, for every possible bit
128     pattern p of n bits,  $p + \text{twosComplement}(p) = 0$ .
129     """
130
131     for p in generateAllBitPatterns(n):
132         # get the two's complement representation for p
133         tp = twosComplement(p)
134
135         # what happens when we add p to tp, ignoring the carry out from the MSB?
136         sum = add(p, tp)
137
138         # display the results for this bit pattern
139         print p, tp, sum
140
141
142 def numberDemo(n):
143     """
144     numberDemo(n) tabulates all possible bit patterns using n bits, showing both
145     unsigned and signed values, assuming the two's complement system is used for
146     signed values.
147     """
148
149     for p in generateAllBitPatterns(n):
150         print p, unsignedToDecimal(p), signedToDecimal(p)
151
152
153 def addDemo(n):
154     """
155     addDemo(n) tabulates all possible pairs of n bit quantities, performs addition
156     on each pair, and summarizes the results, both as bit strings and also as signed
157     decimal values.
158     """
159
160     # Generate all bit patterns of n bits each
161     allBitPatterns = generateAllBitPatterns(n)
162
163     # Generate all pairs of such bit patterns
164     for p in allBitPatterns:
165         for q in allBitPatterns:
166             # Add the pair
167             sum = add(p, q)
168
169             # Convert to signed, decimal values
```

```
170         pD = signedToDecimal(p)
171         qD = signedToDecimal(q)
172         sumD = signedToDecimal(sum)
173
174         # Display a summary
175         print p, "+", q, "=", sum, "or", pD, "+", qD, "=", sumD
176
177
178     def subtractDemo(n):
179         """
180         subtractDemo(n) tabulates all possible pairs of n bit quantities, performs subtraction
181         on each pair, and summarizes the results, both as bit strings and also as signed
182         decimal values.
183         """
184
185         # Generate all bit patterns of n bits each
186         allBitPatterns = generateAllBitPatterns(n)
187
188         # Generate all pairs of such bit patterns
189         for p in allBitPatterns:
190             for q in allBitPatterns:
191                 # Perform the subtraction
192                 diff = subtract(p, q)
193
194                 # Convert to signed, decimal values
195                 pD = signedToDecimal(p)
196                 qD = signedToDecimal(q)
197                 diffD = signedToDecimal(diff)
198
199                 # Display a summary
200                 print p, "-", q, "=", diff, "or", pD, "-", qD, "=", diffD
```