C200 Programming Assignment № 11

Dr. M.M. Dalkilic
Computer Science
School of Informatics, Computing, and Engineering

Indiana University, Bloomington, IN, USA

December 9, 2019

Contents

Introduction

In this homework, you'll work on translating critical thinking to programming. The problems are really fun–so, enjoy! Because this is the last assignment, some of the specifications for output are not presented–you'll need to follow the directions closely. Please start this assignment as soon as possible.

- Add a new folder to your C200 folder named Assignment11. In this folder you'll have the following Python files:
  - **striangle.py**
  - **bestgraph.py**
  - **fullbfs.py**
  - **spaceship.py**
  - **huffman.py**
  - **codd.py**
- Make sure and commit your project and modules by **11pm, Friday December 13th 2019**.

As always, all the work should be your own. You will complete this before 11pm December 13th 2019. You will submit your work by committing your code to your GitHub repository. You will *not* turn anything in on canvas. If your timestamp is 11:01P or greater, the homework cannot be graded. So do not wait until 10:58P to commit your solution. During the last week of classes, the servers tend to become much slower–for safety's sake, submit often. As always, all the work should be your own!

Problem 1: Sierpinkski Triangle

The Polish mathematician Seirpinski is associated with a fractal pattern of an equilateral triangle that is recursively drawn smaller within the larger triangle. One is shown below, the output of the program you'll write. Here is the simple algorithm:

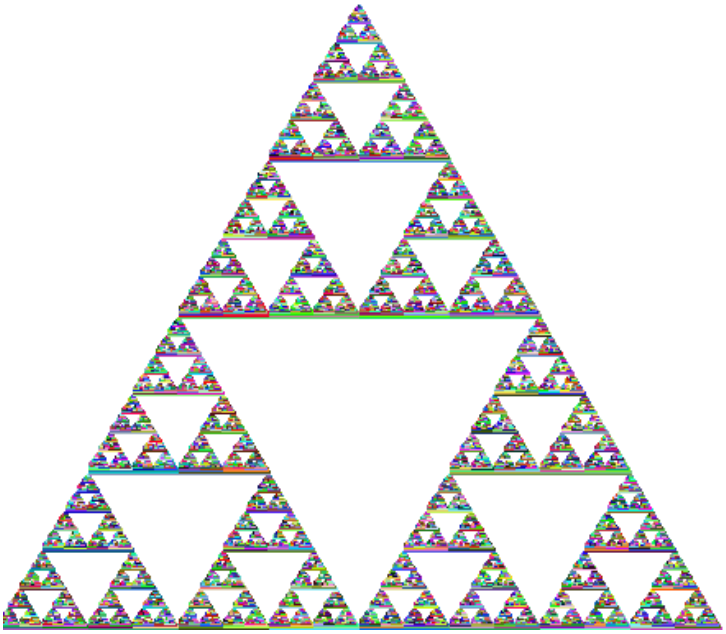

Figure 1: A Sierpinkski Triangle. We've used random colors as we recursively draw smaller triangles. This is what gives it the psychodelic look!

```
#def s(loc, width):
#   if width > 1:
#       Draw a triangle at loc using width
#       s(top, width/2)
#       s(L, width/2)
#       s(R, width/2)
#   else:
#       return
#
# L,R are 1/2 distance on the original sides
```

Look at the following illustration (Fig. 3). The upper-left image is the first triangle drawn. We can specify it completely by giving it the top coordinate and width, because it's an equilateral triangle. We make it easy and assume we want it to be as far left and up as possible. The next step is to draw three smaller triangles (width is now half as much whose tops are pointed to by the yellow arrows in the illustration on the right. We can easily find these values, again, because this is an equilateral triangle. The third illustration is shows the recursion going deeper– we continue drawing triangles yielding the final triangle (this time we used only two colors) to the right.

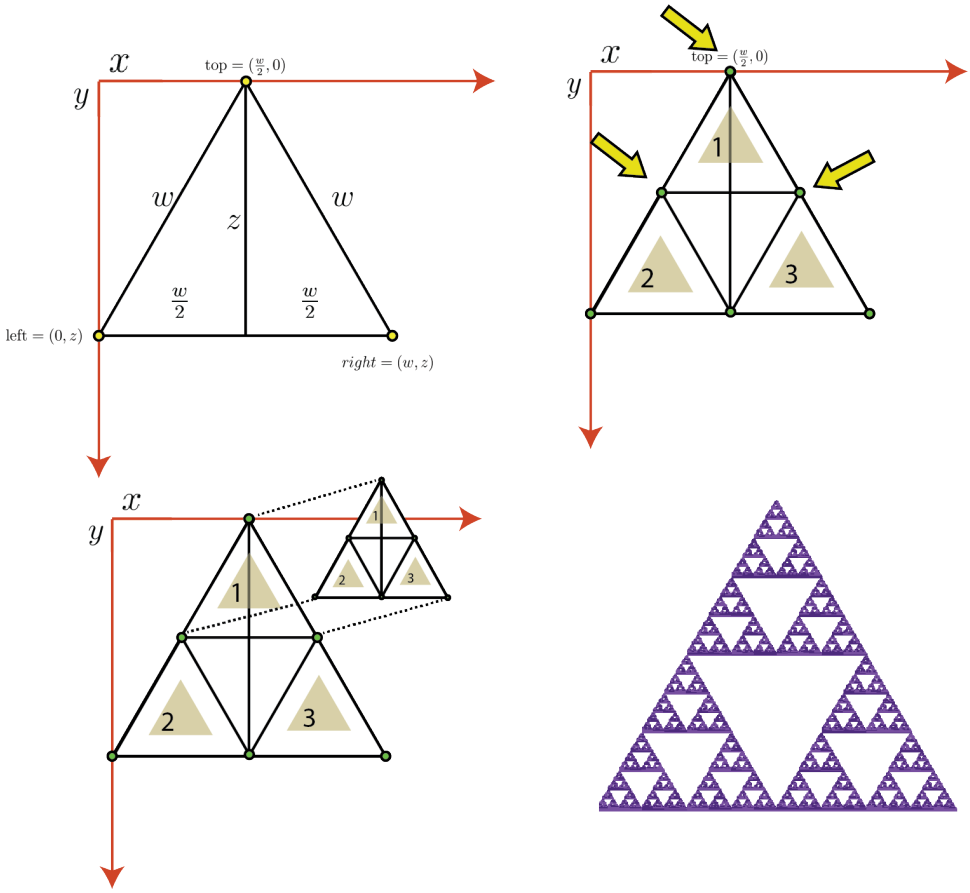

Figure 2: Process of drawing the triangles. We redraw triangles with half the width using the original top, and the midpoints of the left and right sides. The image is the output of the pgoram.

We can use pygame and Pythagorean's Theorem to write a Python program that help us to deteremine the tops (locations of the triangles).

We need to find z where w is some arbitrary width.

$$(\frac{w}{2})^2 + z^2 \quad = \quad w^2 \tag{1}$$

We use `pygame.draw.polygon` to draw a triangle. We need to give a surface, a color, three points, and a non-zero value to indicate we don't want it to fill. You'll need to only complete the recursive function and triangle function.

Listing 1: striangle.py

```
1  import pygame, sys
2  import math
3  from pygame.locals import *
4  import random as rn
5  pygame.init()
6
7  BLUE = (0,0,255)
8  WHITE = (255,255,255)
9
10 DISPLAYSURF = pygame.display.set_mode((500, 400), 0, 32)
11
12 pygame.display.set_caption('S-Triangle')
13
14 #INPUT takes a location loc = (x,y) pair of points and width
15 #RETURN 3 points of the equilateral triangle determined by loc and width
16 def triangle(loc,width):
17     pass
18     #TODO: Implement function
19
20 DISPLAYSURF.fill(WHITE)
21
22 #Draws Triangle
23 #(triangle(loc,w)) is a tuple of tuples...)
24 def draw_triangle(loc, w):
25     pygame.draw.polygon(DISPLAYSURF, BLUE , (triangle(loc,w)),1)
26
27 #INPUT location and width
28 #RETURN nothing -- follows algorithm
29 def s(loc,width):
30     pass
31     #TODO: Implement Function
32
33 s((0,0),440)
34
35 while True:
36     for event in pygame.event.get():
37          if event.type == QUIT:
38             pygame.quit()
39             sys.exit()
40     pygame.display.update()
```

Programming Problem 1: Siepinkski Triangle

- The starting code is in the file **triangle.txt**.
- My advice is to FIRST draw a single triangle – don't begin with the recursion outright.
- Complete the functions above. Your output should look like the ones here or shown on the web.
- For extra credit, display the triangle with random colors throughout the recursion.
- Put your code into a new module named **striangle.py**

Problem 2: Graph Class

In class we implemented a graph class. To instantiate the graph, we had to know the nodes *a priori*. We didn't have a way to manage the graph either–to delete information. We also learned how to use linear algebra to find when one node is reachable from another.

```
class Graph:
    def __init__(self,nodes):
        self.nodes = nodes
        self.edges = {}
        for i in self.nodes:
            self.edges[i] = []
    def add_edge(self, pair):
        start,end = pair
        self.edges[start].append(end)
    def children(self,node):
        return self.edges[node]
    def nodes(self):
        return str(self.nodes)
    def __str__(self):
        return str(self.edges)
```

Here's a small example to reinforce the concept.

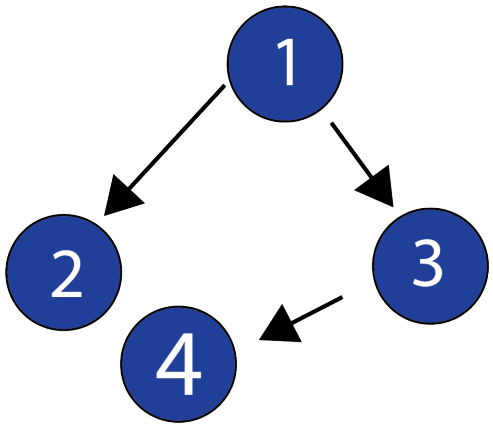

Figure 3: A graph of order 4.

```
#This is an example
import numpy as np
a = np.zeros ((4,4),dtype = int)
a[0][1] = 1
a[1][2] = 1
a[2][3] = 1
print(a)
a = np.dot (a,a) + a
```

```
print(a)
a = np.dot (a,a) + a
print(a)

for i in range(0,4):
    for j in range(0,4):
        if not i == j:
            print("{0} reaches {1}: {2}".format(i+1,j+1,bool(a[i][j])))
```

Session Output 1

```
[[0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]
 [0 0 0 0]]
[[0 1 1 0]
 [0 0 1 1]
 [0 0 0 1]
 [0 0 0 0]]
[[0 1 2 2]
 [0 0 1 2]
 [0 0 0 1]
 [0 0 0 0]]
1 reaches 2: True
1 reaches 3: True
1 reaches 4: True
2 reaches 1: False
2 reaches 3: True
2 reaches 4: True
3 reaches 1: False
3 reaches 2: False
3 reaches 4: True
4 reaches 1: False
4 reaches 2: False
4 reaches 3: False
```

Change

- Allow the user to add an edge (x,y) when x and y are currently in the graph `add_edge((x,y))`. Return 1 if successful and -1 if the edge already exists or either node does not exist.
- Add a method `add_node(n)` that simply adds a new node. Return 1 if successful and -1 if the node already exists.
- Add a method `del_node(n)` that deletes a node and any edges associated with it. Return 1 if successful and -1 if no node existed to delete
- Add a method `del_edge((x,y))` that deletes the edge. Return 1 if successful and -1 if no edge existed to delete.
- Add a method `reachable(x,y)` that returns True if there exists a path from node x to node y, and False if no path exists. Use the transitive closure operation to determine this. You can use the numpy dot method or use your own matrix multiplication function you previously wrote.
- Put your code in a module named **bestgraph.py**
- We have provided a small amount of test code.
- At this point, don't worry about the size of the graphs–they won't be more than 20 or so nodes.

Problem 3: BFS

For the following problem, use your graph class from the previous problem. In class we studied BFS. Although we discussed the algorithm, we didn't provde the code. In the graph (Figure 4),

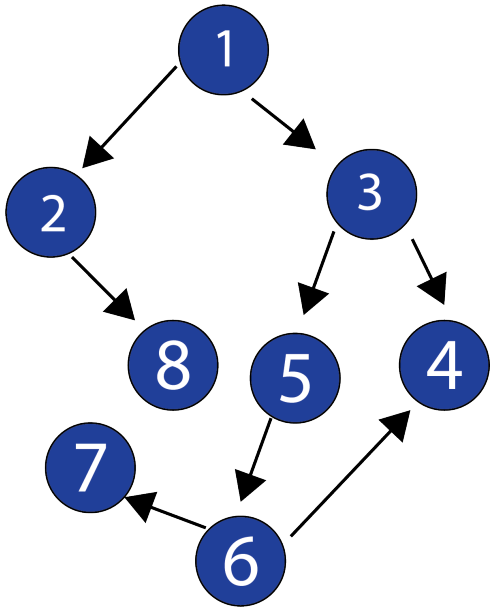

Figure 4: A graph of order 8.

the BFS will visit all nodes if starting with 1; however, any other node will not lead to all nodes being visited. BFS actually keeps track of not only visited nodes, but unvisited. Unvisited nodes are the nodes that are in the graph, but the BFS doesn't visit them when the visited doesn't change.

For example, $u = \{1, 2, 3, 4, 5, 6, 7, 8\}$ (u is unvisited). We do a BFS(G, 5). We'll see: 5,6,7,4 or 5,6,4,7 (either one). If $v = \{5, 6, 7, 4\}$, then we must look at $u - v = \{1, 2, 3, 8\}$. You can use the BFS shown in lecture as a starting point–you must obviously include a queue class (you may copy the one from lab).

```
1  def bfsfull(g,node):
2      #TODO: Implement function
3
4  if __name__=="__main__":
5      my_graph = Graph([1,2,3,4,5,6,7,8])
6      elst = [(1,2),(1,3),(2,8),(3,5),(3,4),(5,6),(6,4),(6,7)]
7      for i in elst:
8          my_graph.add_edge(i)
9      print(my_graph.edges)
10     bfsfull(my_graph,5)
```

Session Output

```
{1: [2, 3], 2: [8], 3: [5, 4],
 4: [], 5: [6], 6: [4, 7], 7: [], 8: []}
5
6
4
7
1
2
3
8
```

We start at 5; 6 is the only child. Then 4,7. We pick randomly 1; 2,3 are children and 8 last.

Programming Problem 3: BFS

- Complete the function.
- BFS does not return anything, it prints when it finds an unvisited node
- Put your code for this problem in a new module named **fullbfs.py**.

Problem 4: Mandelbrot Sets

Some of this material will be familiar, but revisiting it again will be useful! Fig. 5 illustrates a

Figure 5: A Mandelbrot set.

Mandlebrot set. It's a fractal, like the triangle, but it's more like itself. To make them we need complex numbers. A complex number is one that involves the square root of a negative number. sqrt(1) == +(-) 1, but sqrt(-1) can't be done using the square root function we're all familiar with, because no real number squared is -1. Squaring a number multiplies it by itself, but only a positive and a negative number multiplied give a negative number, right? And a positive number could never be the same as a negative number, so we can't calculate the square root! Try it in Python with math.sqrt(), does it work? We need a variable that represents the solutions, or roots, of sqrt(-1), and Python, following Engineering convention, settled on "j" and "-j", while mathematics uses "i", but that is a character reserved for electrical current in this context.

The Complex Plane

A complex number z is represented by c +dj, where c and d are real numbers. The first term is the "real part" and the second, dj, is the "imaginary part". Each can be represented by a number line, and we have them perpendicular, just like the x-and-y-axes system for Euclidean geometry, for convenience. This allows us to write z = (c,d), a tuple of coordinates in the 2-D plane that looks like a point in the 2-D Euclidean plane. With these coordinates we can describe

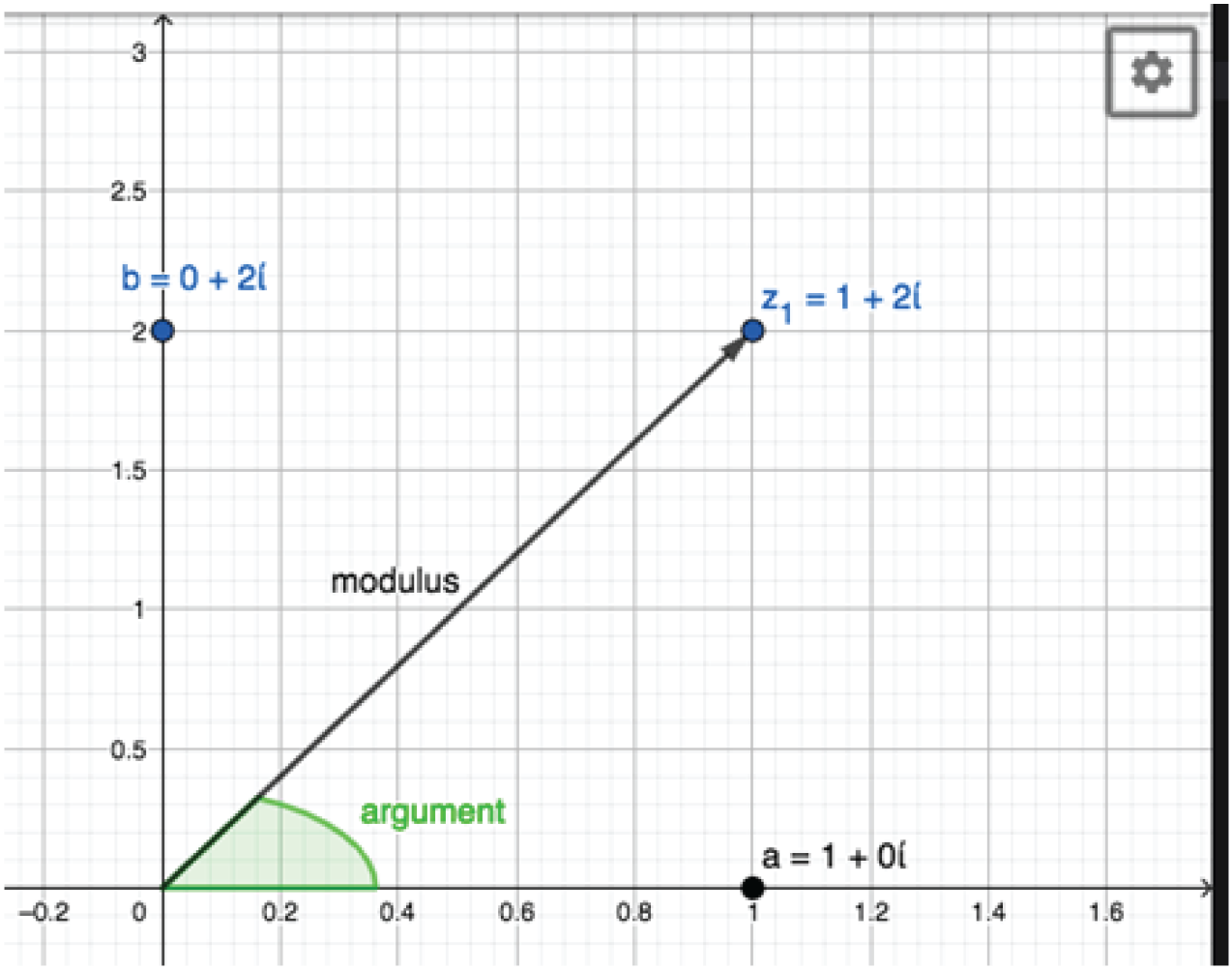

Figure 6: How to plot a complex number.

all complex numbers. This is shown in the complex (argand) plane in Fig. 7. For calculations, you'll need the modulus, which is the distance from a point to the origin, (0,0). Fortunately, the modulus can be calculated the same way the length of the hypotenuse (longest) of a right triangle is in the Pythagorean Theorem using the real coefficients 'c' and 'd' of z = c + dj as the legs. But the similarities between these two number systems are few, for example we need to use a different square root function than the one for positive real numbers, but we won't need that this assignment. Addition and multiplication are also different:

$$e = a + bj \tag{2}$$

$$f = c + dj \tag{3}$$

$$\tag{4}$$

then:

$$e + f = (a + c) + (b + d)j \tag{5}$$

$$e \times f = (ac - bd) + (ad + bc)j \tag{6}$$

The Mandelbrot Equation is:

$$z_0 = 0 \tag{7}$$

$$z_n = z_{n-1}^2 + c \tag{8}$$

where z and c are complex numbers (represented by a tuple for our calculations), and c is the point we want to calculate. A very simple recurrence!

Programming Problem 4

- You cannot use the complex or cmath classes
- Define the function `mandelbrot()` that takes a complex number **represented** as a tuple of coefficients, c, and a maximum number of iterations. It returns the number of iterations needed to leave the circle of radius two. If the maximum is reached, return -1.
- To plot the set, use pygame. The color of a pixel corresponds to a color in a repeating sequence of colors (in the usual depiction). Make your own sequence of colors. In this case a black region represents the Mandelbrot set, which are z values that have not become 'unbounded' (turn your file in using maximum iteration number 60). For each iteration number we want to use a color different from the one or ones next to it. Pick a few and arrange them how you like. How can we pick using an ever increasing index with a small number of colors? Hmm.
- Use an 900x600 pygame window to render the Mandelbrot values of the Real axis from -2 to 1 and Imaginary axis from -1 to 1. Only compute the Mandelbrot values for visible pixels. How many values will we need? How will we store them? Hint: the top left corner is (0,0). You should start with a smaller image for testing. Show the image after drawing everything. Finally, we're ready to navigate. There are so many structures to explore. You could spend a lifetime and never see them all.
- Put your code for this problem in a new module named **spaceship.py**

Problem 5: Huffman encoding and Entropy

Computer networks encode their data into bits, send the bits, then decode the bits into messages. Some data is sent often, and so should use fewer bits to represent them, while other data is rare, and, conversely use more bits. Huffman encoding is a way to arrive at a good encoding based upon the relative presence of the data. Assume we have these data and their counts and respective frequencies:

Data	Count	Frequency
w	7	0.097
u	12	0.167
x	15	0.208
y	18	0.250
z	20	0.278

We then perform this algorithm:

```
1     while there are at least two nodes:
2         find the two smallest counts x, y
3         create a new node with the count x+y
4              and annotate x with 0, y with 1
5
6         This will produce a graph (tree) T.
7         Using T, create a dictionary of each letter.
```

Let's look at the output before the input:

Output

```
[['w', 7], ['u', 12], ['x', 15], ['v', 18], ['y', 20]]
[['x', 15], ['v', 18], [19, ['0', ['w', 7]], ['1', ['u', 12]]],
 ['y', 20]]
[[19, ['0', ['w', 7]],
       ['1', ['u', 12]]], ['y', 20],
 [33, ['0', ['x', 15]],
       ['1', ['v', 18]]]]
[[33, ['0', ['x', 15]],
       ['1', ['v', 18]]],
 [39, ['0', [19, ['0', ['w', 7]],
       ['1', ['u', 12]]]], ['1', ['y', 20]]]]
[[72, ['0', [33, ['0', ['x', 15]], ['1', ['v', 18]]]],
       ['1', [39, ['0', [19, ['0', ['w', 7]],
                  ['1', ['u', 12]]]], ['1', ['y', 20]]]]]]

{'x': '00', 'v': '01', 'w': '100', 'u': '101', 'y': '11'}
```

First, I've edited the lists so they'll fit into the box–so your output will be across the screen (the lists at the end are long). The last two lines are the most important: the penultimate line is the encoding that is in Fig. 7. (F). The last line is the dictionary you've built from the Huffman. You can read more here: `https://en.wikipedia.org/wiki/Huffman_coding`

huffman.py

```
1  #INPUT huffman tree, current code taken from edges
2  #OUTPUT build dictionary of words and code
3  #NOTHING is explicitly returned, it should fill in the dictionary d
4  #RECURSIVE FUNCTION
5  def make_code(xlst,code):
6  #TO DO:IMPLEMENT
7
8  #INPUT list of word, count pairs
9  #OUTPUT huffman tree with a single node
10 #[node [0 or 1 [node]]]
11 def make_tree(xlst):
12 #TO DO: IMPLEMENT FUNCTION
13
14 ###DATA
15 xlst = [['w',7],['u',12],['x',15],['v',18],['y',20]]
16 d = {}
17 f = lambda x: x[1] if type(x[0]) == str else x[0]
18 xlst.sort(key=f) #sorts either original or new nodes
19 print(xlst)
```

```
20
21 make_tree(xlst)
22 make_code(xlst[0],"")
23
24 print(d)
```

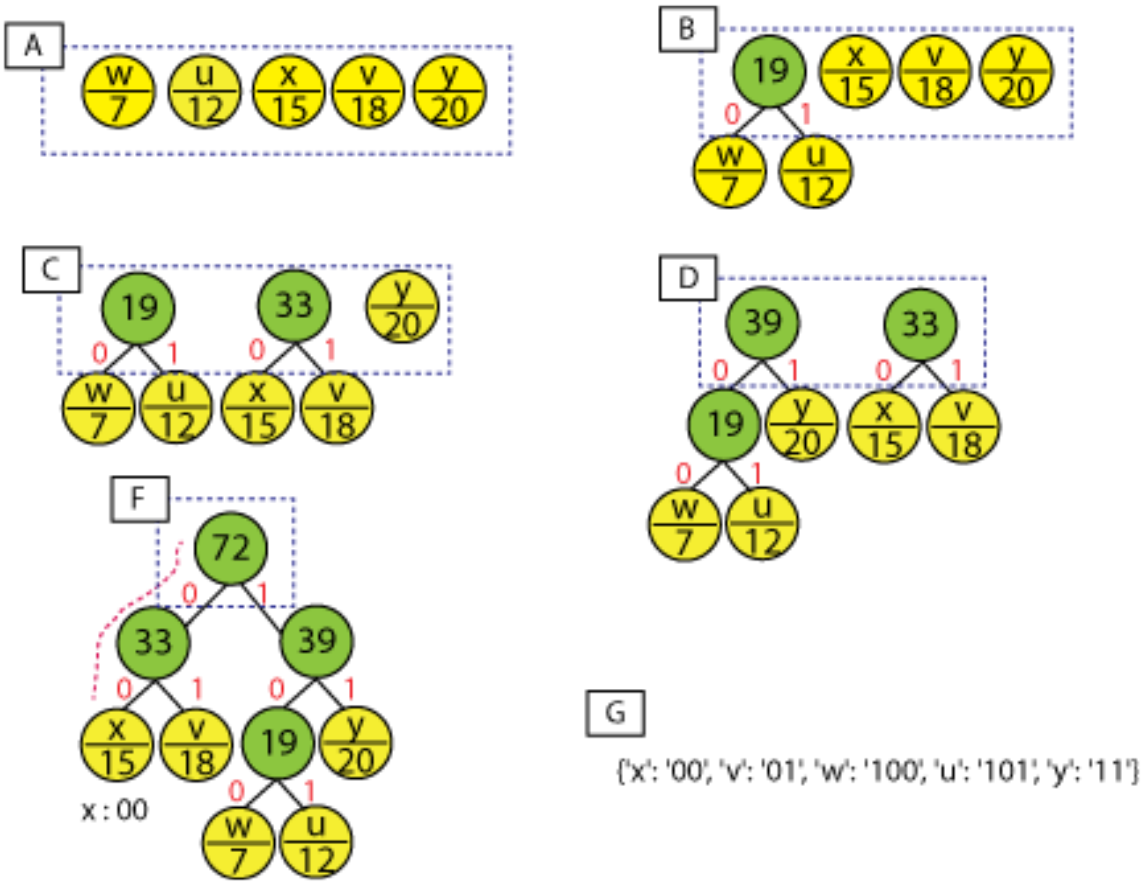

Figure 7: The Huffman Encoding starting at A. The top of the circle is the letter and bottom is the count. B-F show how the algorithm works. In F, the encoding for x, for example, is the path from the top to x using the labels that yields 00. B combines the two smallest counts and creates a node with that count, adds 0,1 to edges. C shows that combining x and v yields the smallest count 33 and this is now added as a subtree. Similarly until we reach F. Then G is the dictionary we build from the tree.*Sorting will help make this problem must easier.*

Programming Problem 5

- Complete the two Huffman functions that produce the dictionary.
- Put your code for this problem in a new module named **huffman.py**

Problem 6: Queries

In class we were introduced to SQL and the relational model. We saw code that built a table from a tree. When running the code below *for the first time*, a database will be created with a single table. Look at

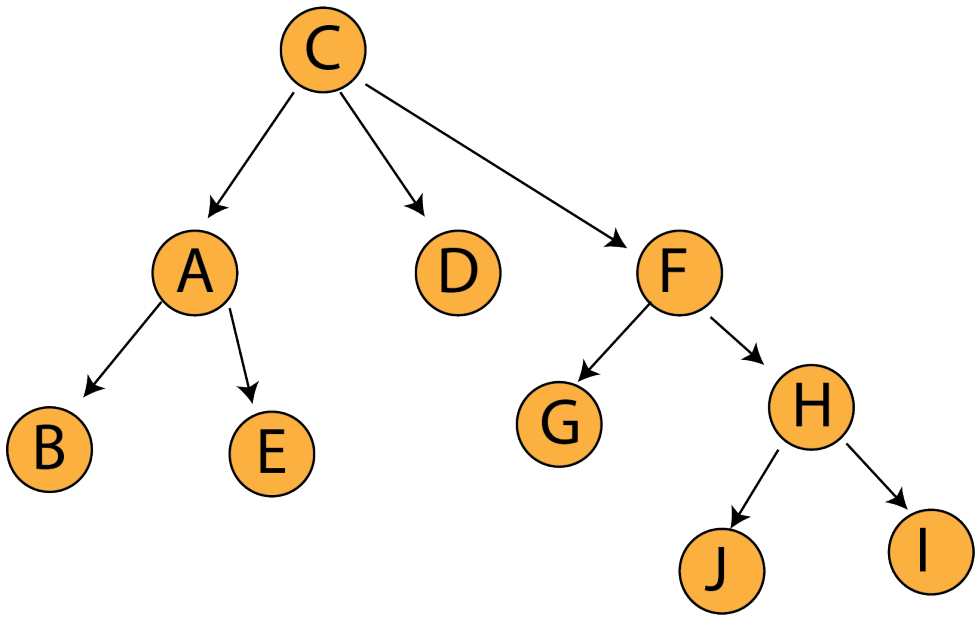

Figure 8: A family tree of letters.

line 16. This is a lambda expression that takes two nodes x and y and asks whether they have the same parent. The query returns the node that is the parent. For example, 'b' and 'e' have the same parent 'a'. 'h' and 'g' share the parent 'f'. However, 'a' and 'e' do not have the same parent.

```
1  import sqlite3
2
3  dog = sqlite3.connect("example1.db")
4  c = dog.cursor()
5  r = [('c','a'),('a','b'), ('a','e'),('c','d'),
6            ('c','f'),('f','g'), ('f','h'),('h','j'),
7            ('h','i')]
8
9  c.execute('DROP TABLE IF EXISTS G')
10 c.execute('''CREATE TABLE G (Parent text, Child)''')
11
12 c.executemany('INSERT INTO G VALUES (?,?)', r)
13
14 dog.commit()
15
16
17 q = #TO DO: Implement Lambda function
18
19 print(c.execute(q('b','e')).fetchone())
20 print(c.execute(q('h','g')).fetchone())
21 print(c.execute(q('e','a')).fetchone())
```

```
22
23 dog.close()
```

Output

```
('a',)
('f',)
None
```

Programming Problem 6

- Complete the lambda expression that allows you to query if two nodes share the same parent.
- Put your code for this problem in a new module named **codd.py**