CSCE 221 Cover Page
Homework Assignment #3
Due August 4 at 23:59 pm to eCampus

First Name Jonathan Last Name Westerfield UIN 224005649

User Name jgwesterfield E-mail address jgwesterfield@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more on Aggie Honor System Office website.

Type of sources	Peer Teacher	Textbook		
People	Lauren Kleckner			
Web pages (provide URL)				
Printed material		Assigned Textbook for the class		
Other Sources				

I certify that I have listed all the sources that I used to develop the solutions/codes to the submitted work.
On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work.

Your Name Jonathan Westerfield Date 8/1/2017

Homework 3 (120 points)

due August 4 at 11:59 pm.

Write clearly and give full explanations to solutions for all the problems. Show all steps of your work.

Reading assignment.

- Priority Queue and Heap, Chap. 8
- Hash Tables and Maps, Chap. 9
- Graphs, Chap. 13

Problems.

1. (10 points) R-9.7 p. 417

   Draw the 11-entry hash table that results from using the has function, $h(k) = (3k + 5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.

Index	Elements	Elements	Elements
0	13		
1	94	39	
2			
3			
4			
5	44	88	11
6			
7			
8	12	23	
9	16	5	
10	20		

2. (10 points) R-9.10 p. 417

   What is the result of Exercise R-9.7, when collisions are handled by double hashing using the secondary hash function $h_s(k) = 7 - (k \bmod 7)$?

Index	Elements
0	13
1	94
2	23
3	88
4	39
5	44
6	11
7	5
8	12
9	16
10	20

3. (10 points) R-8.7 p. 361

An airport is developing a computer simulation of air-traffic control that handles events such as landings and takeoffs. Each event has a *time-stamp* that denotes the time when the event occurs. The simulation program needs to efficiently perform the following two fundamental operations:

- Insert an event with a given time-stamp (that is, add a future event)
- Extract the event with smallest time-stamp (that is, determine the next event to process)

Which data structure should be used for the above operations? Why? Provide big-oh asymptotic notation for each operation.

(a) We use a heap to do this. This is because a min Heap will always have the minimum (smallest) time stamp located at the top of the heap. This gives inserting an element a runtime of $O(logn)$. If we just wanted to see the value of the smallest time stamp, the runtime complextiy would be $O(1)$ because the value would be at the top of the tree. However, for extracting the smallest time stamp and removing that node from the tree, the runtime of this operation would be $O(logn)$. This is because after we get the minimum value, we take it out of the tree and have to restructure the tree.

4. (10 points) R-12.14 p. 588

Draw the frequency array. Use the minimum priority queue based on sorted array to build the Huffman tree for the string below. What is the code for each character and the compression ratio for this algorithm?

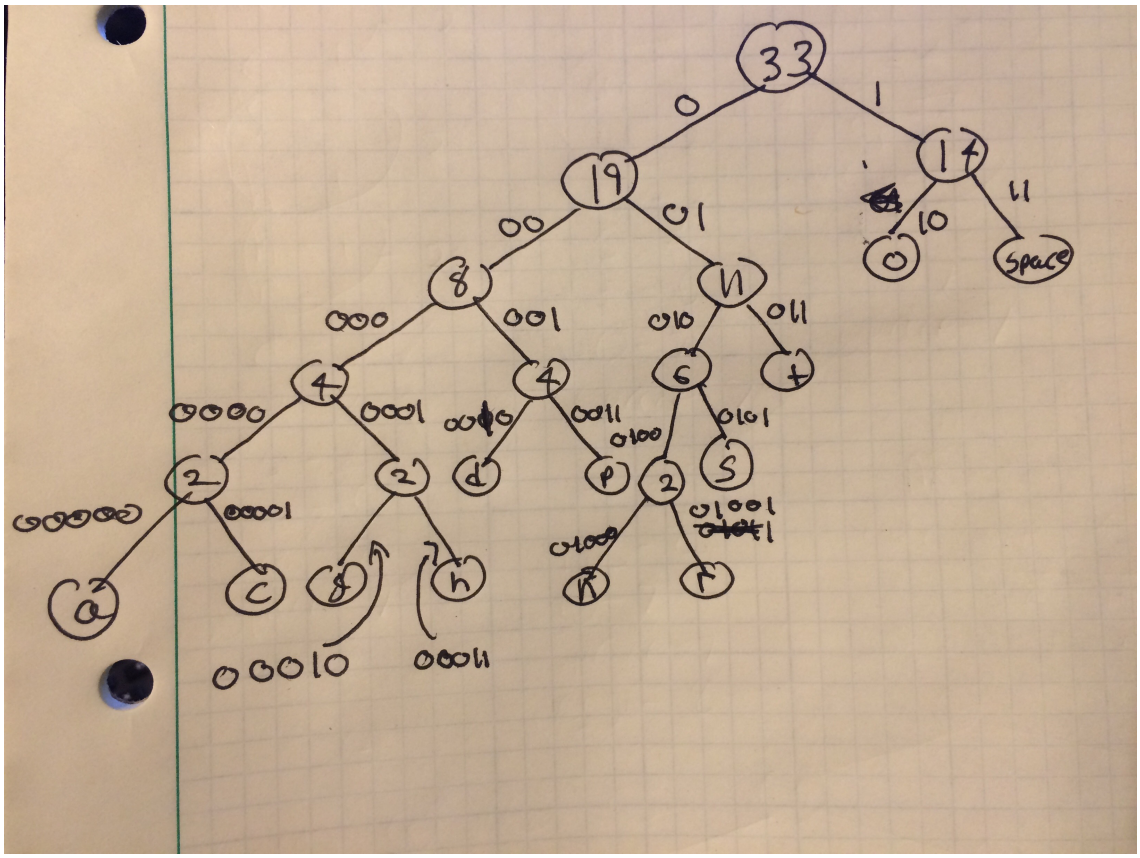
"dogs do not spot hot pots or cats".

(a) Frequency Array:

Character	a	c	g	h	n	r	d	p	s	t	o	space
Count	1	1	1	1	1	1	2	2	4	5	7	7

(b) Code for each character:

i.



Character	A	C	G	H	N	R	D	P	S	T	O	Space
Count	1	1	1	1	1	1	2	2	4	5	7	7
Code	00000	00001	00010	00011	01000	01001	0010	0011	0101	011	10	11

(c) The original length of the string in ASCII code would have made the string 264 bits long. With the Huffman compression, our Huffman code comes to:

i. 00101000010010111001010110100100111101010011100111100011100111100111001101011110010011110000100000

ii. The length of this code is 105 bits long, giving us a compression ratrio of about 35:88 or a compression of about 39.7%.

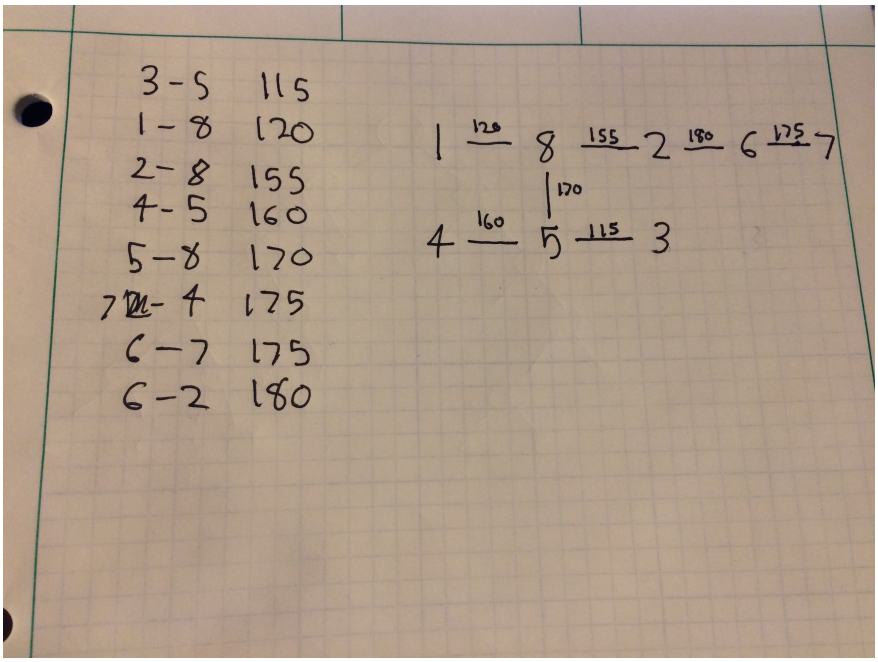
5. (10 points) R-13.15, p. 656

```
Algorith ShortestPath(G, v)
{
    Initialize D[v]=0 and D[u] +inf for each vertex u != v
    while Q is not empty do
    {
        // pull a new vertex u in
        u=Q.removeMin()
    }

    if there is an edge from u to z, then check the distance to z and update is lesser distance is found.
    if(edgeExists(u,z))
    // apply relzation procedure on the edge (u,z)
    if(D[u] + w((u,z)) < D[z]) then
    {
        D[z] = D[u] + w((u,z))
    }

    change to D[z] the key of vertex z in Q.
    return the label of D[u] of each vertex u
}

Algorithm edgeExists(u,z)
{
    if and edge exists between a vertex u and vertex z, the element value at matrix[u][z] is 1 else 0.

    if(matrix[u][z] == 1)
    {
        return true
    }
    else
    {
        return false
    }
}
```

6. (10 points) R-13.16, p. 656

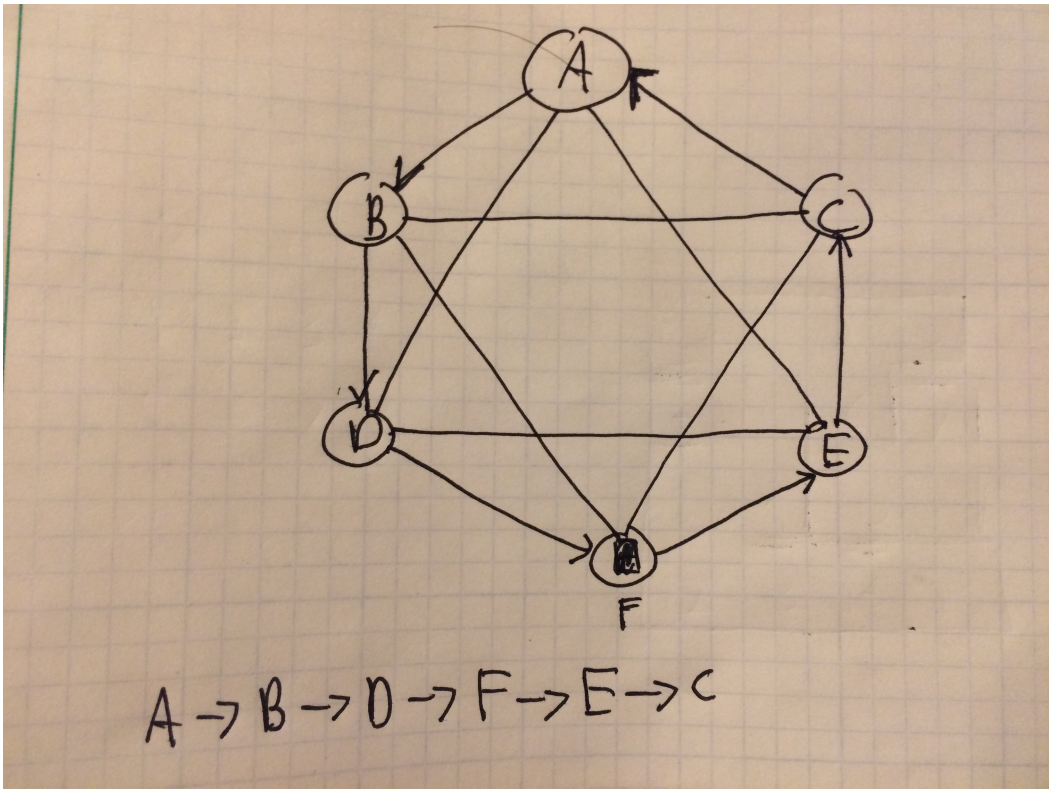
```
Algorithm ShortestPath(G,v)
{
    // the priority queue Que contains all vertices of the graph G using keys as the D labels.

    while Que is not empty do:
    {
        // pull a new vetex u into the cloud
        u = Que.removeMin()
    }

    for each vertex z next to u such that z in Que do:
    {
        as the distance matrix D[z] is updated, display the distance and the path
        // apply relaxation procedure on the edge (u,z)

        if(D[u] + w((u,z)) < D[z]) then:
        {
        D[z] = D[u] + w((u,z))

        print Distance ffrom v to u is dist[i]

        print Path from v to u is v -> z -> u
        }
    }
}
```

7. (10 points) R-13.17, p. 656

(a) 

8. (10 points) R-13.31, p. 657

   (a) The algorithm begins from vertex A, discovers its neighbor B, followed by B's neighbor D and so on. The resulting depth-first search tree of the complete graph is a single sided tree as shown before my drawn graph.

   i. 

9. (10 points) C-13.10, p. 658

   (a) Using Depth First Search, the dges of the graph will be discovered like this: N1 -> N2 -> N3 -> N4 -> N5. This would complete a path around the graph until there are no more paths to follow. However, if there are still edges of the graph that are untraveled, meaning the number of edges traveled does not equal the number of edges in the graph, then the algorithm backtracks to the nearest node with an edge undiscovered path. From there, we travel that edge until we either run into the same problem again or we reach the end of available edges and the number of traveled edges equals the total number of edges in the graph. This allows the paths previously found to be combined. The graph would look similar to this N1 -> N2 -> N3 -> N4 -> N5 -> N6 -> N7 -> N8. The run time of this algorithm is $O(n + m)$ as each edge is traversed at least once while discovering the path and at most $m$times the algorithm backtracks.

10. (10 points) C-13.15, p. 659

    (a) From the graph below, Dijkstra's alorithm would compute an incorrect distance from S to Z to be 4. It wouldn't track the negative edge from X to Y which would create a distance of 2 from S to Z instead of 4. This is a weakness of Dijkstra's algorithm. A negative weight edge fails when performing Dijkstra's algorithm.

(b)