SCMP 218 Fall 2020 Final Exam Take-Home Part

INSTRUCTIONS: You are allowed to use your textbook, your class materials (including video recordings of classes, and other materials in the Google drive), and your class notes on this exam. You are not allowed to use anything else, either online or print. You are not allowed to receive help from another source, human or non-human. Please submit your solutions (repl.it links to your programs) to me via an email by 1:30 pm EST on Friday, December 18.
You must include a statement in your email that you followed the rules of the exam.
Your exam will not be graded if this statement is missing.

1. (50 pts) **Movie Rating** You have collected a file of movie ratings where each movie is rated from 1 (bad) to 5(excellent). The first line of the file is a number that identifies how many ratings are in the file. Each rating then consists of two lines: the name of the movie followed by the numerical rating from 1 to 5. Here is a sample rating file with 4 unique movies and 7 ratings.

```
7
Harry Potter and the Order of the Phoenix
4
Harry Potter and the Order of the Phoenix
5
The Bourne Ultimatum
3
Harry Potter and the Order of the Phoenix
4
The Bourne Ultimatum
4
Wall-E
4
Glitter
1
```

   Write a program that reads a file in this format, calculates the average rating for each movie, and outputs the average along with the number of reviews. Here is what the output should be for the sample data above.

```
Glitter: 1 review, average of 1.0 / 5
Harry Potter and the Order of the Phoenix: 3 reviews, average of 4.3 / 5
The Bourne Ultimatum: 2 reviews, average of 3.5 / 5
Wall-E: 1 review, average of 4.0 / 5
```

   Use a map (or multiple maps) to calculate the output. Your map(s) should index from (associate) a string representing each movie's name to integers (which may even be a vector of integers) that store the number of reviews for the movie and the sum of the ratings for the movie.

2. (50 pts) Do programming exercises 1,2, and 3 at the end pf Chapter 12 of the textbook (page 727). Use a single program to illustrate the 3 algorithms. Pay attention the following points.

   - Note that most of the algorithms, classes, and functions are already implemented by the textbook and the source code is available in the Google drive.
   - However, the textbook source code does not compile as given due to the template inheritance problem that we discussed in Chapter 5. Apply one of the remedies that we discussed to make the given code work.
   - Use an input file for the program to create a graph and apply the two traversal algorithms. The textbook implementation assumes that the graph is represented as an adjacency list. For example, if you use the input file

     ```
     11
     0 1 5 -999
     1 2 3 5 -999
     2 4 -999
     3 -999
     4 3 -999
     5 6 -999
     6 8 -999
     7 3 8 -999
     8 10 -999
     9 4 7 10 -999
     10 -999
     ```

     then, the output of the traversal algorithms should be

     ```
        Depth First Traversal:    0  1  2  4  3  5  6  8  10  7  9
        Breadth First Traversal:  0  1  5  2  3  6  4  8  10  7  9
     ```

     Here, the first piece of data in the file means this digraph has 11 nodes. The next line means the following are the edges out of the vertex 0: $(0, 1)$ and $(0, 5)$. The last number, -999 on each line, means "the end of the input", or the NULL pointer at the end of a linked list.
   - The program should then ask for another file name to receive a weighted digraph to apply the shortest path algorithm. The format of this file should be like the sample below:

     ```
     5
     0 1 3 4 -999
     1 2 -999
     2 1 -999
     3 1 4 -999
     4 1 2 3 -999

     0 0 0 1 16 3 2 4 3 -999
     1 1 0 2 5 -999
     2 1 3 2 0 -999
     3 1 12 3 0 4 7 -999
     4 1 10 2 4 3 5 4 0 -999
     ```

     Here, the first half of the file is exactly the same as the previous sample file that describes a digraph. The bottom half specifies the weights of the edges. For example, the line `0 0 0 1 16 3 2 4 3 -999` means the weight/distance from vertex 0 to itself is 0, from vertex 0 to vertex 1 is 16, from vertex 0 to vertex 3 is 2, from vertex 0 to vertex 4 is 3. When you run the shortest path algorithm on this sample input, the output should be

     ```
     Source Vertex: 0
     Shortest distance from source to each vertex.
     Vertex  Shortest_Distance
        0           0
        1          10
        2           7
        3           2
        4           3
     ```
   - Let the user enter input files as long as they wish.

3. (75 pts) Please choose one (and only one) of the following problems.

   (a) **Basic Coding Theory (Theory of Error Correcting Codes)** First, some definitions. A *code* (an error correcting code) C of length n over an alphabet R is a subset of R^n, the set of all n-tuples (vectors) of length with entries from R, i.e., $C \subseteq R^n$. An element $\mathbf{c} \in C$ is called a *codeword*. Note the analogy with a human language. A set of valid words in a language constitute a subset of all possible strings of letters from the alphabet. We need a few more definitions before the problem statement.

   - *The Hamming distance* between any two vectors $\mathbf{u} = (u_1, u_2, \ldots, u_n)$ and $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ is defined as $d(\mathbf{u}, \mathbf{v}) = \#\{i : u_i \neq v_i\}$, that is the number of positions in which $\mathbf{u}$ and $\mathbf{v}$ differ from each other. For example, the distance between the ternary vectors $\mathbf{u} = (1, 2, 0, 1)$ and $\mathbf{v} = (0, 2, 1, 1)$ is 2.
   - *The Hamming weight* of a vector $\mathbf{u} = (u_1, u_2, \ldots, u_n)$ is defined as $wt(\mathbf{u}) = \#\{i : u_i \neq 0\}$, that is the number of non-zero components of $\mathbf{u}$. Note that $wt(\mathbf{u}) = d(\mathbf{u}, \mathbf{0})$, where $\mathbf{0} = (0, 0, \ldots, 0)$ is the zero vector.
   - *The minimum distance* of a code C is defined as $\min\{d(\mathbf{u}, \mathbf{v}) : \mathbf{u}, \mathbf{v} \in C, \mathbf{u} \neq \mathbf{v}\}$, i.e., it is the minimum of distances between distinct pairs of codewords. Similarly, *the minimum weight* of a code C is defined as $\min\{wt(\mathbf{u}) : \mathbf{u} \in C, \mathbf{u} \neq \mathbf{0}\}$, i.e., it is the smallest weight among non-zero codewords.
   - Let r be an integer between 0 and n. Let A_r denote the number of codewords of (Hamming) weight r in C. *The weight enumerator* of C is defined to be the sequence $(A_0, A_1, \ldots, A_n)$ where only non-zero terms are included in the list.

   Now, your task: Write a program that asks two positive integers m and n from the user and randomly generates a code C of length n with 100 distinct codewords over the alphabet $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. (Make sure that there are no duplicates in the set of codewords. If $m^n < 100$ then print an appropriate error message and ask new input until $m^n \geq 100$.) The program then computes and prints out

      (a) the minimum distance of C,
      (b) the minimum weight of C,
      (c) the weight enumerator of C.

   Use your favorite data structure to store the codewords, and perform the necessary computations. Let the user repeat the program as long as they want.

   (b) **Huffman Codes** Common encoding of a file uses fixed-length codes, typically 8 bits more recently 16 bits, for each character. However, we can save space by using a variable-length coding scheme with shorter codes for frequently occurring characters and longer codes for those that are rarer. Some compression algorithms work in this manner. No matter which method we use, the coding scheme must be uniquely decodable, that is there must not be ambiguity in determining what character is represented when we try to decode. If the codes are all the same length, this is not a problem. Such a code is automatically decodable. For example, the ASCI for the word `BEAD` is the following binary string: `01000010010001010100000101000100`.

   Consider the characters and binary codes shown in the table below.

A	B	C	D	E	F	G
01	001	11	1001	1000	011	111

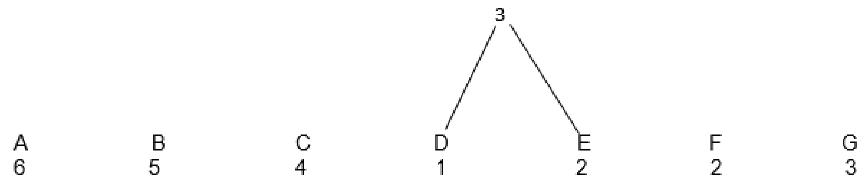
   Using the codes in the table above, the word `BEAD` is represented as `0011000011001`. However, this binary string is ambiguous because it can be decoded both as `BEAD` and as `BEFB`. The problem arises because one code (01 for A) is the prefix of another (011, the code for F). We can eliminate the possibility of ambiguity and also arrive at an encoding that is optimal by employing a technique devised by Huffman (1952) that uses a binary tree. Here is how it works. Assume we have symbols $S_1, \ldots, S_n$ with associated frequencies of $P_1, \ldots, P_n$. As a specific example, consider the symbols and their frequencies in the following table

A	B	C	D	E	F	G
6	5	4	1	2	2	3

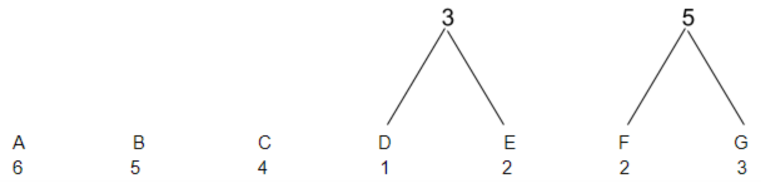
   Huffman's technique for constructing codes works as follows.

```
Make each symbol $S_i$ a single node with associated weight (frequency) $P_i$.
While(more than one parentless node) {
      Identify the two parentless nodes with the smallest weights (breaking ties arbitrarily)
      Create a new node, making it the parent of the two selected nodes,
      and associate with it the sum of the weights of its children. }
Label the branches of the tree:  for each nonleaf node label one of its branches with 0
and the other one with 1.
The code for a symbol is constructed by concatenating the characters (a 0 or a 1) on the path from the
root to the symbol.
```

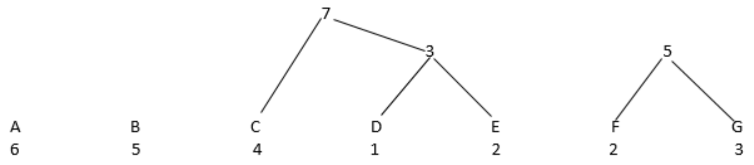
From the data in the last table, we first combine node D (weight 1) with either E or F (weight 2). Choosing E we get the structure shown below.



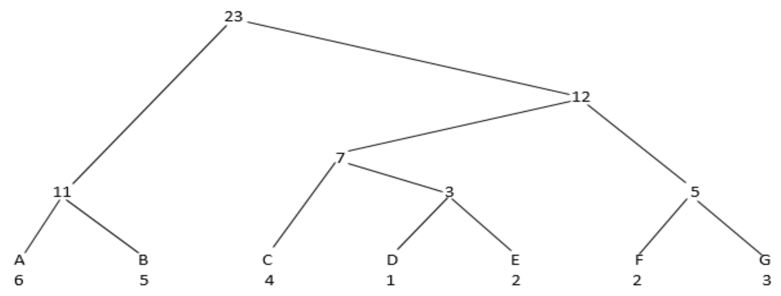
We now combine node F (weight 2) with one of the codes with weight 3. Choosing G gives us the tree below.



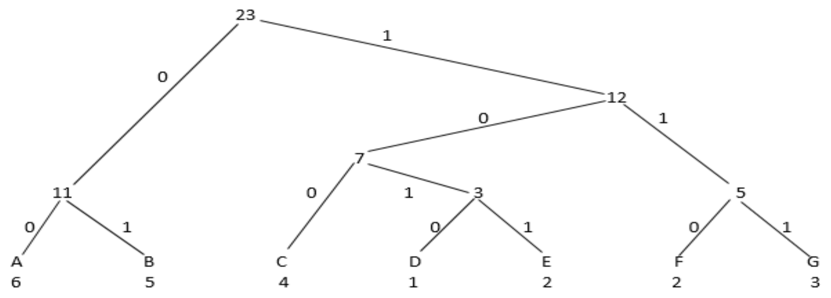
The two nodes with the smallest weights at this point are node C and the node that resulted from our first connection. Joining these yields the structure shown below.



We continue in this manner, finally arriving at the tree shown below (other trees are possible depending on how we break ties).



We now label the links. One of the links from a parent to its child will be labeled 1 and the other will be labeled 0, but it does not matter which is which. Figure below shows a possible tree after this labeling is done.



The codes for each symbol are now derived by following paths from the root to the leaves. The codes for this example are shown below.

A	B	C	D	E	F	G
00	01	100	1010	1011	110	111

According to this table the encoding of `BEAD` is `011011001010`.
To decode a given string, we start at the beginning of the string and with a pointer at the top of the code tree. When reading bits, we follow the appropriate paths down the tree. Whenever we reach a leaf, we output the symbol at that node and reset the pointer to the root of the tree. Try it for yourself using the tree in the last diagram, to verify that `100001101011` decodes to `CAFE`.

Optimality: The average weighted code length is determined using the formula $\frac{\sum P_i \cdot codelength(S_i)}{\sum P_i}$.
For our example, the average weighted code length is $(6*2+5*2+4*3+1*4+2*4+2*3+3*3)/23 \approx 2.652$. It is known that Huffman encoding is optimal in the sense that there is no other prefix-free assignment of codes to symbols that will have a shorter average weighted code length.

Now, your task: Write a program that reads a file that contains the letters of the English alphabet and hypothetical frequencies (weights) of all letters. You can assume each line of the file contains two pieces of data: a letter and its frequency (weight) as a positive integer. Ask the user what the name of the file is. The program then executes the Huffman encoding algorithm and prints i) the encoding of each letter, ii) the average code length for the encoding obtained.