Huffman Compression

Today, we’ll walk through an example of Huffman compression. In Huffman compression, there are three steps:

1. Create a Huffman tree
2. Get all the encodings from our Huffman tree
3. Encoding the text

The text example we’ll work through is “go go gophers”. Then, we’ll have you practice your own example.
Creating the Huffman tree

First, find the frequency of each character in the text we’re trying to encode:

“go go gophers”

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>O</td>
<td>3</td>
</tr>
<tr>
<td>P</td>
<td>1</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>_ (space)</td>
<td>2</td>
</tr>
</tbody>
</table>
Creating the Huffman tree

Then, take the frequency-character pairs and turn them into nodes
Creating the Huffman tree

Then, repeat the following process – take the two smallest-weighted nodes and combine them as follows:

Assign them a parent node that has weight equal to the sum of their weights. Have the one with smaller weight be the left child.

(For the sake of making sure we always generate the same tree – in the case of a tie, pick whatever comes earliest in the alphabet)
Creating the Huffman tree

Once we’ve combined these two nodes, we repeat the process, with this new node as one of our nodes.
Creating the Huffman tree
Creating the Huffman tree
Creating the Huffman tree
Creating the Huffman tree
Creating the Huffman tree

At this point, we’ve connected all our nodes, so we’ve created our Huffman tree!

Now, your turn to practice – try creating a Huffman tree for “abracadabra” – feel free to double check with a partner, but keep in mind there are multiple valid trees!
Huffman Tree

For *abracadabra*

- a: 5
- b: 2
- c: 1
- d: 1
- r: 2

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Compression/Huffman.html
Finding the encoding

To find the encodings – the path from the root of this tree to any character is the encoding, if we replace “left” with 0 and “right” with 1. (Note, at this point we don’t need the frequencies anymore)
Finding the encoding

From this tree, we get the following table of encodings:

Note that characters with higher frequencies have shorter encodings. Now, find the encodings from your tree for “abracadabra” – this property should also be true.

<table>
<thead>
<tr>
<th>Character</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>1100</td>
</tr>
<tr>
<td>G</td>
<td>00</td>
</tr>
<tr>
<td>H</td>
<td>1101</td>
</tr>
<tr>
<td>O</td>
<td>01</td>
</tr>
<tr>
<td>P</td>
<td>1110</td>
</tr>
<tr>
<td>R</td>
<td>1111</td>
</tr>
<tr>
<td>S</td>
<td>100</td>
</tr>
<tr>
<td>_ (space)</td>
<td>101</td>
</tr>
</tbody>
</table>
Encoding the text

Now for the simplest part – to actually encode the text into a bitstream, just replace each character with the encoding:

“go go gophers” → “00 01 101 00 01 101 00 01 1110 1101 1100 1111 100”

<table>
<thead>
<tr>
<th>Character</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>1100</td>
</tr>
<tr>
<td>G</td>
<td>00</td>
</tr>
<tr>
<td>H</td>
<td>1101</td>
</tr>
<tr>
<td>O</td>
<td>01</td>
</tr>
<tr>
<td>P</td>
<td>1110</td>
</tr>
<tr>
<td>R</td>
<td>1111</td>
</tr>
<tr>
<td>S</td>
<td>100</td>
</tr>
<tr>
<td>_ (space)</td>
<td>101</td>
</tr>
</tbody>
</table>
Encoding abracadabra

```
a b r a c a d a b r a
1 011 00 1 0100 1 0101 1 011 00 1
```

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Compression/Huffman.html
Compression visualized

“go go gophers”:
0001101000110100011110110111001111100

“go go gophers” uncompressed:
011001110110111100010000011001110110111100010000011001110110111101110000
01101000011001010111001001110011

“abracadabra”:
1011001010010101100011

“abracadabra” uncompressed:
01100001011000100111001001100001011000110110000101100100011000010110001001110
01001100001
HuffmanDecoding Review

To start thinking about the decoding process for Huffman, we’ll look at the HuffmanDecoding APT.

Note that in HuffmanDecoding the encodings are given to us. In the actual assignment, you’ll have to find the encodings by recreating the tree, using what we’ll call a header, but we’ll delay thinking about that for now.
HuffmanDecoding

The encodings for Huffman have the **prefix property** – no encoding is a prefix of another encoding. So, we know that if “00” is an encoding, “001” can’t be an encoding, or vice-versa. The APT also guarantees this as part of the input.

Our Huffman tree creation process ensures this, because all the nodes representing characters are leaves, and by definition one leaf can’t be on the path to another.
HuffmanDecoding pseudocode

Create a map from Strings to Characters.

For i from 0 to dictionary.length-1:
    Map the i\textsuperscript{th} encoding to ‘A’+I

currentEncoding = ""

Output = ""

For j from 0 to archive.length-1:
    currentEncoding += archive.charAt(j)
    If map has currentEncoding as a key:
        output += map.get(currentEncoding)
        currentEncoding = ""

Return output
Huffman decoding with a tree

Suppose we did not have a map of encodings to characters, but instead a Huffman tree. (Again, we’ll ignore how we actually get the tree from the encoded information for now)

Then, rather than search in our map, we could instead start at the root, and move left or right on our Huffman tree based on what the next bit of our archive is.

If we reach a leaf node, we add the character it represents to our output, and return to the root.

current = root
For j from 0 to archive.length-1:
    currentEncoding += archive.charAt(j)  # Go left or right based on archive.charAt(j)
    If map has currentEncoding as a key: If current is a leaf node:
        output += map.get(currentEncoding)  # current’s character value
        currentEncoding = ""  # current = root
ErdosNumbers: The Problem

http://www.cs.duke.edu/csed/newapt/erdosnumbers.html

Problem Statement

The Erdos number is a way of describing the "collaborative distance" between a scientist and Paul Erdos by authorship of scientific publications.

Paul Erdos is the only person who has an Erdos number equal to zero. To be assigned a finite Erdos number, a scientist must publish a paper in co-authorship with a scientist with a finite Erdos number. The Erdos number of a scientist is the lowest Erdos number of his coauthors + 1. The order of publications and numbers assignment doesn't matter, i.e., after each publication the list of assigned numbers is updated accordingly.

You will be given a String[] pubs, each element of which describes the list of authors of a single publication and is formatted as "AUTHOR_1 AUTHOR_2 ... AUTHOR_N" (quotes for clarity only). Paul Erdos will be given as "ERDOS".

Return the list of Erdos numbers which will be assigned to the authors of the listed publications. Each element of your return should be formatted as "AUTHOR NUMBER" if AUTHOR can be assigned a finite Erdos number, and just "AUTHOR" otherwise. The authors in your return must be ordered lexicographically.

```java
public class ErdosNumber {
    public String[] calculateNumbers(String[] pubs) {
        // you write code here
    }
}
```
Outline

Today in discussion, we’ll go over the general problem solving strategy from the first discussion, except catering it to graph problems:

- Work through an example
- Figure out how we worked through that example
- Figure out how to turn the input into a graph
- Figure out how to solve the problem given the graph
- Write code to turn the input into a graph in Java
- Write code to solve the problem in Java
Examples

{“KLEITMAN LANDER”, “ERDOS KLEITMAN”} → {“ERDOS 0”, “KLEITMAN 1”, “LANDER 2”}


Try working through the following two by hand:

{"ERDOS A", "A B C", "B D E", “C F G”} → ?

{"ERDOS A B", “A B C”, “B C D”, “E F”} → ?
Examples

{“KLEITMAN LANDER”, “ERDOS KLEITMAN”} → {“ERDOS 0”, “KLEITMAN 1”, “LANDER 2”}

Try working through the following two by hand:

One possible (general) strategy

Start with a table of Erdos numbers, with ERDOS = 0. Then, at each step, simultaneously update as many as possible until you reach a step where no updates can be made.

e.g. for \{"ERDOS A B", "A B C", "B C D", "E F\}
Implementation

You could implement the strategy on the previous slide using a double for loop, and it would solve the APT correctly. However, this would run in $O(n^2)$, which might time out the tester (and also avoids the purpose of the exercises, which is graph practice).

Instead, we’ll talk about implementing this strategy by turning the input into a graph.
Turning ErdosNumbers into a graph

Given our input for the ErdosNumbers APT (a series of co-publishers of some number of papers), what should our vertices represent?

What should our edges represent? i.e., given a set of co-publishers, how can we determine the set of edges we want to use?

(This is a good step-by-step approach for turning any problem into a graph problem)

Try drawing the graph for the following set of co-publishers. Also figure out what the adjacency list would look like.

You should have gotten:


Now, using the graph, can you quickly find the ERDOS numbers?
Breadth-first search

Because each vertex’s Erdos number is just the minimum distance from the ERDOS vertex to that vertex, and breadth-first search (BFS) finds these distances efficiently, we’ll implement our solution using BFS.

Recall the following pseudo-code for BFS:

Create a queue of vertices Q
Q.push(start vertex)
While Q is not empty:
    curr = Q.pop
    if we’ve visited curr, continue
    mark curr as visited
    do something at curr
    push all of curr’s neighbors into Q

For this problem, what is the start vertex? What is the “do something” we will do at curr?
Marking nodes as visited

This is our customized version of BFS for the ErdosNumbers problem. All we have left to do is find a way to define which vertices are visited.

```
dist = map of strings to ints
dist[“ERDOS”] = 0
Create a queue of vertices Q
Q.push(“ERDOS”)
While Q is not empty:
    curr = Q.pop
    if we’ve visited curr, continue
    mark curr as visited
    for each of curr’s neighbors:
        if neighbor is not a key in dist:
            dist[neighbor] = dist[curr] + 1
    push all of curr’s neighbors into Q
```
Once we have constructed the graph, the following pseudo-code will correctly determine all distances:

```
dist = map of strings to ints
dist["ERDOS"] = 0
Create a queue of vertices Q
Create a set of visited vertices S
Q.push("ERDOS")
While Q is not empty:
    curr = Q.pop
    if curr in S:
        continue
    S.add(curr)
    for each of curr's neighbors:
        if neighbor is not a key in dist:
            dist[neighbor] = dist[curr] + 1
    push all of curr's neighbors into Q
```
Implementing in Java

Now that we have specified how to do everything (short of turn our distance map into the final output, which should be relatively simple) in a generic sense, let us specify how to do it within Java.

All the data structures we use in BFS on the previous slide (sets, queues, maps) are well-defined within Java, except for the graph itself.

Graphs are most commonly stored as adjacency lists – a data structure which when passed a vertex, returns a list of neighbor vertices.

What would be the best way to represent an adjacency list in Java?
Constructing the graph

Our code for constructing the graph will probably look like this:

```java
HashMap<String, HashSet<String>> myGraph = new HashMap<String, HashSet<String>>();
for(String publication: pubs){
    //split pubs into authors
    //for each author in authors
        //if myGraph does not have author as a key:
            myGraph.put(author, new HashSet<String>());
        //add each other author in authors to myGraph.get(author)
}
```

We can then quickly access all of curr’s neighbors using `myGraph.get(curr)`.

Because we use a HashSet to keep track of neighbors, we will never have duplicate edges in our graph. However, we might sometimes have two authors who co-published multiple papers, which could correspond to duplicate edges. Why would this not affect the correctness of our solution?
A brief note on efficiency

In a graph with $n$ vertices and $m$ edges, BFS runs in $O(n+m)$ – a substantial improvement on the $O(n^2)$ result we would get using a double for loop.

This is because in the case where every vertex is interconnected, $m$ is $O(n^2)$, but when there are fewer edges $m$ is a stricter bound than $n^2$. 
Finish the APT!

At this point we’ve discussed everything you should need to know to finish the APT. Spend the rest of discussion writing your solution.